Chapter 7

μML and type inference

Contents

It soon appeared intolerable to have to declare—for example—a new maplist function for mapping a function over a list, every time a new type of list is to be treated. Even if the maplist function could possess what Strachey called "parametric polymorphism," it also appeared intolerable to have to supply an appropriate type explicitly as a parameter, for each use of this function.

Robin Milner, *How ML Evolved*

Chapter 6 presents the statically typed languages Typed Impcore and Typed μScheme. The monomorphic language Typed Impcore is easy to program in and easy to write a type checker for, but the only way to add polymorphic type constructors and functions is to extend the interpreter. The polymorphic language Typed μScheme is easy to extend with new polymorphic type constructors and functions, and it is easy to write a type checker for, but it is difficult to program in: as Milner observed, adding an explicit type parameter at every use of every polymorphic value soon becomes intolerable. In this chapter, we present a language that offers the best of both worlds: μML. ML is easy to program in and easy to extend with new polymorphic functions. We pay for this ease of use with a more complicated typing algorithm: instead of type checking, μML uses *type inference*.

In a language with type inference, you don't have to write explicit type annotations; instead, the implementation of the language finds the types of variables and parameters. Type inference is used in such languages as Haskell, Hope, Miranda, OCaml, and Standard ML. In order to get type inference, language designers have to give up some expressive power: the type-inference problem for a language as powerful as Typed μScheme is undecidable. To make type inference decidable, we restrict polymorphism. The most popular restriction is the *Hindley-Milner type system*. In this type system, a quantified ∀ type may appear only at top level; a ∀ type may *not* be passed to a type constructor. In particular, a ∀ type may not appear as an argument in a function type.

We present the Hindley-Milner type system and its type inference in the context of μML, a language that is closely related to Typed μScheme.

- Like Typed μScheme, μML has first-class, higher-order functions, and its values are S-expressions.
- Like Typed μScheme, μML has polymorphic types that are checked at compile time.
- Unlike Typed μScheme, μML has *implicit* types. In μML, the programmer never writes a type or a type constructor.
- Unlike Typed μScheme, μML has no mutation. μML lacks `set`, and names stand for values, not for mutable locations. Because there is no mutation, programming in μML is nearly always done in *applicative* style. The exception is input/output, which is done imperatively, with `print`.
- Unlike Typed μScheme, μML puts restrictions on polymorphism: quantified types may appear only at top level. In exchange for this restriction, we get a benefit: μML does type abstraction and instantiation automatically. A μML programmer never has to write `type-lambda` or `@`.

μScheme, Typed μScheme, and μML are very similar. If we erase the types from a Typed μScheme program, we get a valid μScheme program. If the program does not use `set` or `while`, and if it uses `type-lambda` appropriately, it will also be a valid μML program.

Like our interpreter for Typed μScheme, our interpreter for μML is based on the μScheme interpreter from Chapter 5. As with the type checker from Chapter 6, we present the rules for type inference but leave substantial parts of the implementation as Exercises.

7.1 μML: a nearly applicative language

Aside from the type system, the primary difference between μML and μScheme is that μML does not permit mutation. (To add mutation to μML, see Exercise 20.) Mutation is a primary example of an *imperative feature*. Although μML does not have mutation, it does have other imperative features: `print`, `error`, and `begin` (see sidebar).

In μScheme, a `val` definition can mutate an existing binding, but in μML, `val` always creates a new binding. To enable the definition of recursive functions, μML introduces the new definition form `val-rec`, and it uses `define` as syntactic sugar for a combination of `val-rec` and `lambda`. A function defined with `val-rec` or `define` can make recursive calls to itself. A nest of *mutually* recursive functions, however, can be defined only using `letrec`.

Another difference between μML and μScheme is that μML needs fewer primitives. Because μML has a type system, we know at compile time what is a symbol, a number, and a function, so μML doesn't need any of these primitives:

```
boolean? number? symbol? pair? procedure?
```

μML does need the `null?` primitive, which is used to tell the difference between empty and non-empty lists.

Except for the addition of `val-rec`, the concrete syntax of μML is a subset of that of μScheme.

def ::= `(val` *variable-name exp*`)`
| `(val-rec` *variable-name exp*`)`
| *exp*
| `(define` *function-name* `(`*formals*`)` *exp*`)`
| `(use` *file-name*`)`

exp ::= *literal*
| *variable-name*
| `(if` *exp exp exp*`)`
| `(begin` {*exp*}`)`
| `(`*exp* {*exp*}`)`
| `(`*let-keyword* `(`{`(`*variable-name exp*`)`}`)` *exp*`)`
| `(lambda (`*formals*`)` *exp*`)`
| *primitive*

let-keyword ::= `let` | `let*` | `letrec`

formals ::= {*variable-name*}

literal ::= *integer* | `#t` | `#f` | `'`*S-exp* | `(quote` *S-exp*`)`

S-exp ::= *literal* | *symbol-name* | `(`{*S-exp*}`)`

primitive ::= `+` | `-` | `*` | `/` | `=` | `<` | `>` | `print` | `error`
| `car` | `cdr` | `cons` | `null?`
| `pair` | `fst` | `snd`

integer ::= sequence of digits, possibly prefixed with a minus sign

**-name* ::= sequence of characters not an *integer* and not containing `(`, `)`, `;`, or whitespace

Sidebar: Imperative features

The early chapters of this book show a spectrum of designs from imperative to functional. Impcore has only second-class functions, and as in C, most programs will probably be written using `set`, `while`, and `begin`. μScheme and Typed μScheme have the same imperative features as Impcore, but the emphasis is on programming with first-class functions. μML has the same first-class functions as μScheme and Typed μScheme, but it has fewer imperative features. In Chapter 8, μHaskell has no imperative features at all. So what are imperative features, anyway, and why do we care about them?

A simple rule of thumb is that *a feature is imperative if when that feature is used, different orders of evaluation can produce different answers.*[a]

- The `set` construct, also called assignment or mutation, is an imperative feature: if two expressions assign to the same location, the order of evaluation matters. In particular, after two assignments, the mutable location holds the value written by the second assignment.
- Input/output is an imperative feature, because if two different `print` expressions are evaluated in different orders, the program's output is different. Similarly, if a program reads x and y from its input, it may produce different answers depending on which order the variables are read in.
- Exceptions are an imperative feature, because if two different subexpressions could raise exceptions, which exception is raised depends on which subexpression is evaluated first. In μScheme, `error` is a similar imperative feature, because the error message depends on the order of evaluation.

Languages with imperative features usually have other features that are not themselves imperative, but are closely related.

- Sequencing with `begin` is not an imperative feature; its purpose is to enable the programmer to control order of evaluation. In languages such as C, C++, and OCaml, which have imperative features but do not specify order of evaluation of function arguments, sequencing is essential; otherwise, some programs' behavior would be impossible to predict.
- Loops are interesting only in the presence of imperative features. Without imperative features, for example, a `while` expression has one of only three uninteresting outcomes: nontermination, termination, or halting with a run-time error.

Why should we care about imperative features? For two reasons:

- In the absence of imperative features, it is easier to reason about the behavior of programs, especially using algebraic laws. It is therefore easier to construct correct programs and to transform programs to improve their efficiency.
- If imperative features are absent or restricted, a compiler can often produce better code. In ML-like languages, restrictions on mutation have been used to reduce memory requirements, improve instruction scheduling, and reduce the overhead of garbage collection.

A language containing no imperative features may be called *pure*; a language containing imperative features may be called *impure*.

[a] For purposes of deciding whether a feature is imperative, we don't consider nontermination to be a "different answer."

7.2 Abstract syntax and values of μML

As in Chapter 5, we define the abstract syntax of μML by presenting type definitions from our implementation. It is a subset of the abstract syntax of μScheme.

287a ⟨*abstract syntax and values* 287a⟩≡ (331c) 287b▷

```
datatype exp = LITERAL of value
             | VAR     of name
             | IFX     of exp * exp * exp
             | BEGIN   of exp list
             | APPLY   of exp * exp list
             | LETX    of let_kind * (name * exp) list * exp
             | LAMBDA  of name list * exp
and let_kind = LET | LETREC | LETSTAR
and ⟨definition of value 287c⟩
```

No feature in the abstract syntax is imperative, but μML is an impure language because it has two imperative primitives: `print` and `error`. It is therefore useful to have `BEGIN`.

Except for `VALREC`, definitions are exactly as in μScheme.

287b ⟨*abstract syntax and values* 287a⟩+≡ (331c) ◁287a

```
datatype def = VAL    of name * exp
             | VALREC of name * exp
             | EXP    of exp
             | DEFINE of name * (name list * exp)
             | USE    of name
```

At a mathematical level, μML and μScheme have the same values, but in our interpreters, we use subtly different representations. Because μScheme has `set`, environments map names to mutable cells. In μML, environments map names to values. Without mutable locations, how are we to implement recursive functions? The value of a function is a closure, and to make a recursive call possible, the closure must contain the value of the function. To implement a closure that refers to itself in this way, we need some sort of cyclic representation. In our implementation of μScheme, we use mutation to create a graph with a cycle. In our implementation of μML, we build each closure's environment *lazily*; that is, instead of storing an environment in the closure, we store a function that produces an environment on demand.

287c ⟨*definition of* value 287c⟩≡ (287a)

```
value = NIL
      | BOOL      of bool
      | NUM       of int
      | SYM       of name
      | PAIR      of value * value
      | CLOSURE   of lambda * (unit -> value env)
      | PRIMITIVE of primop
withtype primop = value list -> value (* raises RuntimeError *)
     and lambda = name list * exp
exception RuntimeError of string (* error message *)
```

type env 214
type name 214

As in Chapter 5, we use `RuntimeError` to signal errors. Because the values are nearly the same, we reuse the projection, embedding, and printing functions from Chapter 5.

7.3 Operational semantics

Because μML doesn't have mutation, and because we don't formally specify the effects of the imperative primitives `print` and `error`, μML's operational semantics is very simple. The abstract machine has no locations and no store; evaluating an expression produces a value, period. The judgment is $\langle e, \rho \rangle \Downarrow v$. Evaluating a definition produces a new environment; the form of that judgment is $\langle d, \rho \rangle \to \rho'$.

7.3.1 Rules for expressions

I hope that most of the rules are self-explanatory.

$$\frac{}{\langle \textsc{literal}(v), \rho \rangle \Downarrow v} \qquad (\textsc{Literal})$$

$$\frac{x \in \operatorname{dom} \rho}{\langle \textsc{var}(x), \rho \rangle \Downarrow \rho(x)} \qquad (\textsc{Var})$$

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \qquad v_1 \neq \textsc{bool}(\texttt{\#f}) \qquad \langle e_2, \rho \rangle \Downarrow v_2}{\langle \textsc{if}(e_1, e_2, e_3), \rho \rangle \Downarrow v_2} \qquad (\textsc{IfTrue})$$

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \qquad v_1 = \textsc{bool}(\texttt{\#f}) \qquad \langle e_3, \rho \rangle \Downarrow v_3}{\langle \textsc{if}(e_1, e_2, e_3), \rho \rangle \Downarrow v_3} \qquad (\textsc{IfFalse})$$

The rules for BEGIN are a bit of a cheat; the purpose of BEGIN is to force order of evaluation, but our rules are so simplified that they don't enforce an order of evaluation.

$$\frac{}{\langle \textsc{begin}(), \rho \rangle \Downarrow \mathit{nil}} \qquad (\textsc{EmptyBegin})$$

$$\frac{\begin{array}{c}\langle e_1, \rho \rangle \Downarrow v_1 \\ \langle e_2, \rho \rangle \Downarrow v_2 \\ \vdots \\ \langle e_n, \rho \rangle \Downarrow v_n\end{array}}{\langle \textsc{begin}(e_1, e_2, \ldots, e_n), \rho \rangle \Downarrow v_n} \qquad (\textsc{Begin})$$

Just as in μScheme, LAMBDA captures an environment in a closure, and APPLY uses the captured environment, but because no locations are involved, the rules are simpler.

$$\frac{}{\langle \textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle \Downarrow (\!|\textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e), \rho|\!)} \qquad (\textsc{MkClosure})$$

$$\frac{\begin{array}{c}\langle e, \rho \rangle \Downarrow (\!|\textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c|\!) \\ \langle e_1, \rho \rangle \Downarrow v_1 \qquad \cdots \qquad \langle e_n, \rho \rangle \Downarrow v_n \\ \langle e_c, \rho_c\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \rangle \Downarrow v\end{array}}{\langle \textsc{apply}(e, e_1, \ldots, e_n), \rho \rangle \Downarrow v} \qquad (\textsc{ApplyClosure})$$

We take advantage of having functions in our value space; the mathematical rule for applying a μML primitive is to apply the function attached to that primitive. The implementation is equally simple.

$$\frac{\begin{array}{c}\langle e, \rho\rangle \Downarrow \text{PRIMITIVE}(f) \\ \langle e_1, \rho\rangle \Downarrow v_1 \\ \vdots \\ \langle e_n, \rho\rangle \Downarrow v_n\end{array}}{\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho\rangle \Downarrow f([v_1, \ldots, v_n])} \qquad \text{(APPLYPRIMITIVE)}$$

Let-expressions are simplified because we don't have to deal with locations.

$$\frac{\langle e_1, \rho\rangle \Downarrow v_1 \quad \cdots \quad \langle e_n, \rho\rangle \Downarrow v_n \quad \langle e, \rho\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\rangle \Downarrow v}{\langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho\rangle \Downarrow v} \qquad \text{(LET)}$$

As in μScheme, a LETSTAR expression requires a sequence of environments.

$$\frac{\begin{array}{c}\langle e_1, \rho_0\rangle \Downarrow v_1 \qquad \rho_1 = \rho_0\{x_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}\rangle \Downarrow v_n \qquad \rho_n = \rho_{n-1}\{x_n \mapsto v_n\} \\ \langle e, \rho_n\rangle \Downarrow v\end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho_0\rangle \Downarrow v} \qquad \text{(LETSTAR)}$$

LETREC is the tricky one. The expressions have to be evaluated in an environment in which their names are already bound to the resulting values. In μML, as in full ML, we can pull this off if the expressions are all λ abstractions.

$$\frac{\begin{array}{c}e_1, \ldots, e_n \text{ are all LAMBDA expressions} \\ \rho' = \rho\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\ \langle e_1, \rho'\rangle \Downarrow v_1 \quad \cdots \quad \langle e_n, \rho'\rangle \Downarrow v_n \\ \langle e, \rho'\rangle \Downarrow v\end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho\rangle \Downarrow v} \qquad \text{(LETREC)}$$

The restriction that each e_i is a LAMBDA expression makes it easy to construct a ρ' that satisfies the equations in the premises.

7.3.2 Rules for evaluating definitions

In μScheme or Typed μScheme, evaluating a definition can change the store. In μML, because there is no mutation, a definition is evaluated primarily for its effect on the environment.[1] We therefore write the judgment in the form $\langle d, \rho\rangle \to \rho'$.

There are several significant differences between μML and μScheme:

- In μML, as in full ML and in Typed μScheme, a VAL declaration never mutates a previous binding; it always adds a new binding. (See Exercise 37 on page 166 in Chapter 3.) If existing functions or other values depend on the old binding, those functions *refer* to the old binding, not the new one. If you are using an interactive interpreter, and you change one binding but you don't re-enter the definitions of the functions that depend on the old binding, you may be quite puzzled by the subsequent behavior of the interpreter. This problem trips many programmers new to ML, but after experience, most ML programmers view VAL's behavior as an important feature.

$$\frac{\langle e, \rho\rangle \Downarrow v}{\langle \textsc{val}(x, e), \rho\rangle \to \rho\{x \mapsto v\}} \qquad (\textsc{Val})$$

- Like Typed μScheme but unlike μScheme, μML has VAL-REC. A VAL-REC defines a recursive value: given $\textsc{val-rec}(x, e)$, we must evaluate e in an environment that already maps x to the value of e. In Typed μScheme, we solve this problem using mutation. First we create an environment that maps x to a mutable cell holding unspecified contents, then we evaluate e in that environment, and finally we store the result in the cell. In μML, because environments map directly to values, we can't map x to a mutable cell, and there is no cell to update after we evaluate e.

  Fortunately, we need VAL-REC only to define recursive functions at top level. Given the definition $\textsc{val-rec}(f, \textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e))$, we can produce a closure for $\textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e)$ such that the closure contains a function that builds the proper environment on demand. In a formal definition, we can simply require a ρ' that binds f to a closure containing ρ', without worrying about how it might be implemented.

$$\frac{\rho' = \rho\{f \mapsto (\!|\textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e), \rho'|\!)\}}{\langle \textsc{val-rec}(f, \textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e)), \rho\rangle \to \rho'} \qquad (\textsc{ValRec})$$

- In μML, as in Typed μScheme, the definition $\textsc{define}(f, a, e)$ is syntactic sugar for $\textsc{val-rec}(f, \textsc{lambda}(a, e))$.

$$\frac{\langle \textsc{val-rec}(f, \textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e)), \rho\rangle \to \rho'}{\langle \textsc{define}(f, \langle x_1, \ldots, x_n\rangle, e), \rho\rangle \to \rho'} \qquad (\textsc{Define})$$

The rule for expressions is just like the rule in μScheme; evaluating a top-level expression e is syntactic sugar for evaluating a binding to `it`.

$$\frac{\langle \textsc{val}(\texttt{it}, e), \rho\rangle \to \rho'}{\langle \textsc{exp}(e), \rho\rangle \to \rho'} \qquad (\textsc{Exp})$$

[1] Evaluating a definition may also print.

7.4 Type system for μML

Just as in Typed Impcore and Typed μScheme, the type system of μML rejects programs that might commit certain faults at run time. (Because you can't write a type explicitly, μML's type system is no use as documentation.) The type system determines which terms have a type, and the implementation accepts a definition only if its terms have types. As before, we specify which terms have types by using a formal proof system. Before getting into the details of the proof rules, I discuss the elements of the system.

7.4.1 Types, type schemes, and type environments

Just as in Typed μScheme, we build types using four elements:

- Type variables, which we write using α
- Type constructors, which we write generically using μ or specifically using a name such as `int` or `list`
- Constructor application, which we write using the ML notation $(\tau_1, \ldots, \tau_n)\ \tau$
- Quantification, which we write using $\forall$

The difference between Typed μScheme and μML is that types in μML are restricted: a type quantified with $\forall$ may appear only at top level, never as an argument to a type constructor. In μML, we build this restriction into the *syntax* of types.

- The symbol τ stands for a type built with type variables, type constructors, and constructor application:

  $$\tau ::= \alpha \mid \mu \mid (\tau_1, \ldots, \tau_n)\ \tau$$

  A τ is called a *type*.

- The symbol σ stands for a type quantified over a list of type variables $\alpha_1, \ldots, \alpha_n$:

  $$\sigma ::= \forall \alpha_1, \ldots, \alpha_n\ .\ \tau$$

  A σ is called a *type scheme*.

The difference between types and type schemes is reflected in the abstract syntax:

291 ⟨*Hindley-Milner types* 291⟩≡ 294a▷

```
datatype ty = TYVAR  of name                (* type variable alpha *)
            | TYCON  of name                (* type constructor mu *)
            | CONAPP of ty * ty list        (* type-level application *)
```

type name 214

```
datatype type_scheme = FORALL of name list * ty
```

The set of type schemes σ in which the $\alpha_1, \ldots, \alpha_n$ is empty is isomorphic to the set of types τ. The isomorphism relates each type `tau` to the type scheme `FORALL([], tau)`. A type without $\forall$ or a type scheme in which the $\alpha_1, \ldots, \alpha_n$ is empty is sometimes called a *monotype* or *ground type*. A type scheme in which the $\alpha_1, \ldots, \alpha_n$ is not empty is sometimes called a *polytype*.

When I write a type application in code, I put the type constructor *before* its arguments, as in

```
CONAPP(TYCON "list", [TYCON "int"]).
```

But when I write a type application in the text, I put the type constructor *after* its arguments, as in `int list`. The notation I use in the text is consistent with ML notation.

In μML, a number of these type constructors are predefined. The essential ones are the `tuple` constructor and the `function` constructor, since they are used in the rules for functions. Other constructors, like `bool`, `int`, `sym`, and so on are needed to give types to literals or primitives.

We use the ML abbreviations for tuple and function types:

Type	*Abbreviation*
(τ_1, τ_2) `function`	$\tau_1 \to \tau_2$
$(\tau_1, \ldots, \tau_n)$ `tuple`	$\tau_1 \times \tau_2 \cdots \times \tau_n$
() `tuple`	`unit`
$(\tau_1)\ \tau$	$\tau_1\ \tau$

Another difference between μML and Typed μScheme is that in μML, kinds are unnecessary. The reason is that in μML, a programmer never writes an explicit type—therefore it is impossible for a programmer to write an ill-kinded type. In full ML, where a programmer can write types, kinds are useful.

In μML, as in Typed μScheme, we write a type environment using the Greek letter Γ. In μML, a type environment Γ maps a term variable[2] to a type scheme. The type environment is used only during type inference; it is not needed at run time.

7.4.2 Substitution, instances, and instantiation

The great virtue of the μML type system is that *the system automatically instantiates polymorphic values.* As an example, we define `nil` with type $\forall\alpha\,.\,\alpha$ `list`, then instantiate it at types `int` and `sym list`:

292 ⟨*transcript* 292⟩≡ 298▷

```
-> (val nil '())
() : forall 'a . 'a list
-> (val p (pair (cons 1 nil) (cons '(a b c) nil)))
((1) ((a b c))) : int list * sym list list
```

In Typed μScheme, we would have had to use the explicit instantiations `(@ nil int)` and `(@ nil (list sym))`. We would also have had to instantiate `pair` and `cons` explicitly.[3] An explicit instantiation or type application substitutes types written by the programmer for the quantified type variables of a polymorphic type. In ML, it is up to the implementation to figure out what types to substitute. Milner's type inference figures out the types by computing an appropriate *substitution.* Because substitutions play a much greater role in ML than in Typed μScheme, we represent them explicitly.

cons 𝒫 329b
pair 𝒫 806c

[2] Term variables, which appear in terms (expressions) and are bound by `let` or `lambda`, stand for values. They are not to be confused with type variables, which stand for types. In full ML, the name of a term variable begins with a letter; the name of a type variable begins with the prime (') symbol.

[3] These instantiations are what Professor Milner found intolerable.

A substitution is a finite map from type variables to types. We write a substitution using the Greek letter θ (pronounced "THAYT-uh"). In our explanation of the Hindley-Milner type system, we interpret a substitution in many different ways:

- As a function from type variables to types
- As a function from types to types
- As a function from type schemes to type schemes
- As a function from type environments to type environments
- As a function from type-equality constraints to type-equality constraints
- As a function from typing judgments to typing judgments
- As a function from typing derivations to typing derivations

We use all these interpretations in the math, and we use some of them in the code. The interpretation we use most is the function from types to types. Such a function θ is a substitution if it preserves type constructors and constructor application as follows:

- For any type constructor μ, $\theta\mu = \mu$.
- For any constructor application $\text{CONAPP}(\tau, \langle\tau_1, \ldots, \tau_n\rangle)$,

  $$\theta(\text{CONAPP}(\tau, \langle\tau_1, \ldots, \tau_n\rangle)) = \text{CONAPP}(\theta\tau, \langle\theta\tau_1, \ldots, \theta\tau_n\rangle).$$

  Or, using informal ML-like notation,

  $$\theta((\tau_1, \ldots, \tau_n)\ \tau) = (\theta\tau_1, \ldots, \theta\tau_n)\ (\theta\tau).$$

  In the common case where the τ being applied is a simple constructor μ, $\theta\mu = \mu$ and

  $$\theta((\tau_1, \ldots, \tau_n)\ \mu) = (\theta\tau_1, \ldots, \theta\tau_n)\ \mu.$$

To be a substitution, a function from types to types must meet one other condition:

- The set $\{\alpha \mid \theta\alpha \neq \alpha\}$ must be *finite*. This set is the set of variables substituted for. It is called the *domain* of the substitution, and it is written $\operatorname{dom}\theta$.

We use substitution to define exactly when τ' is an *instance* of τ: we write $\tau' <: \tau$ if and only if there exists a substitution θ such that $\tau' = \theta\tau$. The relation $\tau' <: \tau$ is pronounced in two ways: not only "τ' is an *instance* of τ" but also "τ is *more general* than τ'."

We extend the instance relation to type schemes as follows:

> We write $\tau' <: \forall\alpha_1, \ldots, \alpha_n.\tau$ if and only if there exists a substitution θ such that $\operatorname{dom}\theta \subseteq \{\alpha_1, \ldots, \alpha_n\}$ and $\tau' = \theta\tau$. The first condition says that instantiating substitution θ may substitute only for type variables that are bound by the $\forall$.

To *instantiate* a type scheme $\sigma = \forall\alpha_1, \ldots, \alpha_n\,.\,\tau$ is to choose a $\tau' <: \sigma$. When we do this, the definition of $<:$ permits us to substitute for the type variables $\alpha_1, \ldots, \alpha_n$, and we are permitted to substitute *only* for those variables. The big deal about ML is that the system instantiates each σ automatically.

Of all the ways we can interpret a substitution, the one I've chosen for the *representation* of a substitution is a finite map from type variables to types. In our code, we have a standard representation for such a map: an environment of type `ty env`. To interpret a substitution as a function from type variables to types, we apply `varsubst` to it:

294a ⟨*Hindley-Milner types* 291⟩+≡ ◁291 294b▷

```
type subst = ty env
fun varsubst theta =
  (fn a => find (a, theta) handle NotFound _ => TYVAR a)
```

```
type subst
varsubst : subst -> (name -> ty)
```

As the code shows, the function defined by a substitution is *total*. If a type variable is not in the domain of the substitution, the function leaves that variable unchanged.

The most frequently used interpretation of a substitution is as a function from types to types. That interpretation is provided by function `tysubst`. The code is almost exactly the same as the code in Chapter 6.

294b ⟨*Hindley-Milner types* 291⟩+≡ ◁294a 294c▷

```
fun tysubst theta =
  let fun subst (TYVAR a) = varsubst theta a
        | subst (TYCON c) = TYCON c
        | subst (CONAPP (tau, taus)) = CONAPP (subst tau, map subst taus)
  in  subst
  end
```

```
tysubst : subst -> (ty -> ty)
subst   :          ty -> ty
```

A function produced by `tysubst` has type `ty -> ty` and so can be composed with any other function of the same type, including all functions that correspond to substitutions. To be precise, if θ_1 and θ_2 are substitutions, then `tysubst` $\theta_2 \circ$ `tysubst` θ_1 is a function from types to types (and also corresponds to a substitution). Composition is really useful, but a substitution *data structure* θ is strictly more useful than the corresponding *function* `tysubst` θ. For one thing, we can interrogate θ about its domain. To have the best of both worlds, I define a function for composing substitutions, which obeys the algebraic law

$$\texttt{tysubst}(\texttt{compose}(\theta_2, \theta_1)) = \texttt{tysubst}\ \theta_2 \circ \texttt{tysubst}\ \theta_1.$$

bindList 214
BindListLength 214
CONAPP 291
emptyEnv 214
type env 214
find 214
FORALL 291
NotFound 214
type ty 291
TYCON 291
TYVAR 291
union 318a

Function `dom` produces a substitution's domain.

⟨*Hindley-Milner types* 291⟩+≡ ◁294b 294d▷

```
⟨sets of free type variables 318a⟩
fun dom theta = map (fn (a, _) => a) theta
fun compose (theta2, theta1) =
  let val domain  = union (dom theta2, dom theta1)
      val replace = tysubst theta2 o varsubst theta1
  in  map (fn a => (a, replace a)) domain
  end
```

```
dom     : subst -> name set
compose : subst * subst -> subst
```

Instantiation is just as in Chapter 6, except no kind environment is needed. Because the system ensures everything has the right kind, it is an internal error to instantiate with the wrong number of arguments. The internal error is signalled by raising the exception `BugInTypeInference`. This exception is raised only when there is a fault in the interpreter; a faulty μML program should never trigger this exception.

294d ⟨*Hindley-Milner types* 291⟩+≡ ◁294c 295a▷

```
exception BugInTypeInference of string

fun instantiate (FORALL (formals, tau), actuals) =
  tysubst (bindList (formals, actuals, emptyEnv)) tau
  handle BindListLength => raise BugInTypeInference "number of types in instantiation"
```

```
instantiate : type_scheme * ty list -> ty
```

I finish this section with three more definitions related to substitutions. These definitions provide the empty substitution `idsubst` as well as two functions used to create and change substitutions. The substitution that maps every type variable to itself is sometimes called the empty substitution (because its domain is empty) and sometimes the identity substitution (because when viewed as a function from types to types, it is the identity function). In code it is `idsubst`; in math it is θ_I, and it obeys the algebraic law $\theta_I \circ \theta = \theta \circ \theta_I = \theta$.

295a ⟨*Hindley-Milner types* 291⟩+≡ ◁294d 295b▷

```
val idsubst = emptyEnv
```

`idsubst : subst`

The simplest substitutions are those that substitute a single type for a single variable. To make it easy to create such a substitution, I define a new infix operator `|-->`. The expression `alpha |--> tau` is the substitution that substitutes `tau` for `alpha`. In math, we write that substitution $(\alpha \mapsto \tau)$.

295b ⟨*Hindley-Milner types* 291⟩+≡ ◁295a 295c▷

```
infix 7 |-->
val idsubst = emptyEnv
fun a |--> (TYVAR a') = if a = a' then idsubst else bind (a, TYVAR a', emptyEnv)
  | a |--> tau        = if member a (freetyvars tau) then
                          raise BugInTypeInference "non-idempotent substitution"
                        else
                          bind (a, tau, emptyEnv)
```

`|--> : name * ty -> subst`

The `|-->` function doesn't accept just any α and τ. It produces a substitution $(\alpha \mapsto \tau)$ only when type variable α is *not* free in τ. It's a little hard to motivate this limitation, but the idea is to ensure that the `|-->` function produces only *idempotent* substitutions. An idempotent substitution θ has the property that

$$\theta \circ \theta = \theta.$$

Idempotence by itself is not so interesting, but if $\theta = (\alpha \mapsto \tau)$ is idempotent, then we are guaranteed the following equality between types:

$$\theta\alpha = \theta\tau.$$

Because type inference is all about using substitutions to guarantee equality of types, we want to be sure that every substitution we create is idempotent. (An example of a substitution that is *not* idempotent is $(\alpha \mapsto \alpha$ `list`$)$.)

bind 214
BugInType-Inference 294d
emptyEnv 214
FORALL 291
freetyvars 318b
member 318a
typeString 693
TYVAR 291

7.4.3 Other functions to manipulate types

Appendix H defines a function `typeString`, which we use to print types. We also use this function to print type schemes, but when we print a true polytype, we make the `forall` explicit, and we show all the quantified variables.[4]

295c ⟨*Hindley-Milner types* 291⟩+≡ ◁295b 296a▷

```
⟨printing types (generated automatically)⟩
fun typeSchemeString (FORALL ([], ty)) =
      typeString ty
  | typeSchemeString (FORALL (a::a's, ty)) =
      let fun combine (var, vars) = vars ^ ", '" ^ var
      in  "forall " ^ foldl combine ("'" ^ a) a's ^ " . " ^ typeString ty
      end
```

`typeString : ty -> string`
`typeSchemeString : type_scheme -> string`

[4] It is not strictly necessary to show the quantified variables, because in any top-level type, *all* type variables are quantified by the ∀. For this reason, Standard ML leaves out quantifiers and type variables. But when you're learning about parametric polymorphism, it's better to make the `forall`s explicit.

Because there are no quantifiers in a type, the definition of type equality is simpler than the corresponding definition for Typed μScheme in chunk 267a.

296a ⟨*Hindley-Milner types* 291⟩+≡ ◁295c 296b▷

```
fun eqType (TYCON c, TYCON c') = c = c'
  | eqType (CONAPP (tau, taus), CONAPP (tau', taus')) =
      eqType (tau, tau') andalso eqTypes (taus, taus')
  | eqType (TYVAR a, TYVAR a') = a = a'
  | eqType _ = false
and eqTypes (t::taus, t'::taus') = eqType (t, t') andalso eqTypes (taus, taus')
  | eqTypes ([], []) = true
  | eqTypes _ = false
```

eqType : ty * ty -> bool

To make it easier to define the primitive operations of μML, I provide convenience functions very much like those from Chapter 6.

296b ⟨*Hindley-Milner types* 291⟩+≡ ◁296a 319a▷

```
val inttype  = TYCON "int"
val booltype = TYCON "bool"
val symtype  = TYCON "sym"
val alpha    = TYVAR "a"
val beta     = TYVAR "b"
fun tupletype taus  = CONAPP (TYCON "tuple", taus)
fun pairtype (x, y) = tupletype [x, y]
val unittype        = tupletype []
fun listtype ty     = CONAPP (TYCON "list", [ty])
fun funtype (args, result) =
  CONAPP (TYCON "function", [tupletype args, result])
```

```
inttype   : ty
booltype  : ty
symtype   : ty
unittype  : ty
listtype  : ty -> ty
tupletype : ty list -> ty
pairtype  : ty * ty -> ty
funtype   : ty list * ty -> ty
alpha     : ty
beta      : ty
```

7.4.4 Type rules for μML

In Typed μScheme, you are free to define and use a type in which the quantifier ∀ may appear anywhere. You pay for this freedom by writing `type-lambda` and `@` everywhere. The examples in Chapter 6 show just how unpleasant it is to write these type abstractions and type applications explicitly. ML makes polymorphism lightweight and efficient by dispensing with explicit type abstraction and type application; the price is that the language is more restricted. In ML, the ∀ quantifier appears only in a type scheme, never in a type, and it is impossible to write a function that expects an argument of ∀ type.

Because ML programs don't have explicit types and specialization, when we're checking the type of a μML program, we have to work harder than we did to check Typed μScheme.

CONAPP 291
TYCON 291
TYVAR 291

- When we see the definition of a function, the types of the arguments aren't given. We have to figure out the most general type we can for each argument.
- When we see an expression containing a polymorphic value, the expression doesn't say how to instantiate the value. We have to figure out the type at which the value should be instantiated.

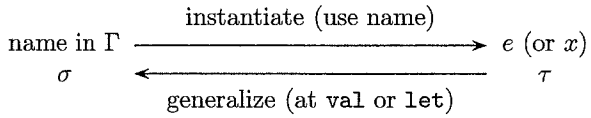

Figure 7.1: Relationship between type schemes and types

The restriction on $\forall$ makes it possible to solve these problems directly in the type system. A key property of the ML type system is that *all expressions have monotypes*, although those monotypes may have free type variables. *Only variables bound in the environment may have polytypes.* When we take a name from Γ and use it as an expression, we *instantiate* its type scheme to give it a monotype. This instantiation amounts to an implicit @ operation. When we process a `val` or `let` binding, we take the type of the expression and *generalize* it to make a type scheme, which we then put into the environment. This generalization amounts to an implicit `type-lambda` operation. For a simple visualization of what's going on, see Figure 7.1. (There's also a sidebar on page 323.)

When we instantiate a type, how do we know what types to substitute for the type variables? When we determine the type of a function, how do we know what types to use for the arguments? Luckily, we don't have to answer these questions right away. We can write *nondeterministic* rules such that a μML program is well typed if it can be assigned a type by these rules. Nondeterministic rules make great specifications, but translating them to an algorithm is less obvious than in a type-checked language with deterministic rules. The rest of this section presents the type rules; Sections 7.5.1 and 7.5.2 explain how to get from the nondeterministic rules to a deterministic type-inference algorithm.

Type rules for expressions

The typing judgment for an expression is $\Gamma \vdash e : \tau$, meaning that given type environment Γ, expression e can be used at type τ. Because of polymorphism, the type of an expression is not unique; for example, in the μML type system, the judgments $\Gamma \vdash$ LITERAL(NIL) : `int list` and $\Gamma \vdash$ LITERAL(NIL) : `bool list` are both valid.

The use of a variable is well typed if the variable is bound in the environment. We must instantiate the variable's type scheme.

$$\frac{\Gamma(x) = \sigma \qquad \tau' <: \sigma}{\Gamma \vdash x : \tau'} \qquad \text{(VAR)}$$

Unlike our rules for operational semantics, this rule does *not* specify a deterministic algorithm. *Any* τ' that is an instance of σ is acceptable, and it is not immediately obvious how to come up with the "right" one. We have more to say about this problem in Section 7.5.2; our implementation comes up with a *most general* τ' (see also Exercise 3 on page 334).

A conditional expression is well typed if the condition is Boolean and the two arms have the same type τ. The type of the conditional expression is also τ.

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau} \qquad \text{(IF)}$$

Because sequential composition is used to execute expressions for their side effects, we would be justified in requiring all expressions except the last to have type `unit`. But because full ML allows expressions in a sequence to have any type, we do also.

$$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n}{\Gamma \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n} \qquad \text{(BEGIN)}$$

The premises mentioning $e_1, \ldots, e_{n-1}$ are necessary because although we don't care what the types of $e_1, \ldots, e_{n-1}$ are, each one still has to be well typed.

We give the empty BEGIN type `unit`.

$$\frac{}{\Gamma \vdash \text{BEGIN}() : \texttt{unit}} \qquad \text{(EMPTYBEGIN)}$$

A function has an arrow type, and only a value of arrow type can be applied to other values. The types of the actual parameters must match the types of the formal parameters.

$$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \qquad \Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau} \qquad \text{(APPLY)}$$

To compute the type of a function we have to know the types of its arguments. These give us the type of the body, which then determines the type of the function.

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e) : \tau_1 \times \cdots \times \tau_n \to \tau} \qquad \text{(LAMBDA)}$$

The LAMBDA rule is the other nondeterministic rule; given a LAMBDA, it is not immediately obvious how to choose $\tau_1, \ldots, \tau_n$. In the premise, the notation $\{x_i \mapsto \tau_i\}$ is shorthand for $\{x_i \mapsto \forall.\tau_i\}$; that is, each τ_i is converted into a type scheme by wrapping it in an empty $\forall$. The type scheme $\forall.\tau_i$ has only one instance, which is τ_i itself. Therefore, when x_i is used in e, it always has the same type.

So a LAMBDA-bound variable has a type with an empty $\forall$. So what? How does this property affect a programmer? The property implies that an ML programmer cannot define a function that *requires* its arguments to be polymorphic. No matter how polymorphic the *actual* parameter may be, inside the function the *formal* parameter has just one type. This property is a key part of the Hindley-Milner type system. (It is needed to make type inference decidable.)

For example, although we can treat the global variable `nil` as polymorphic, consing both an integer and a boolean onto it, we cannot define a function that does the same thing with a formal parameter:

cons 𝒫 329b
pair 𝒫 806c

⟨*transcript* 292⟩+≡ ◁292 299▷

```
-> (val nil '())
() : forall 'a . 'a list
-> (val p (pair (cons 1 nil) (cons #t nil)))
((1) (#t)) : int list * bool list
-> (val too-poly (lambda (nil) (pair (cons 1 nil) (cons #t nil))))
type error: cannot make int equal to bool
```

The example shows the difference between a `nil` bound by `lambda` and a `nil` bound by `val`. Because `val` is the definition form that corresponds to `let`, this difference is also called the difference between a lambda-bound variable and a let-bound variable. To illustrate this difference, here is the typing rule for MLET, a restricted form of LET that binds a single variable.

$$\frac{\Gamma \vdash e' : \tau' \qquad \text{ftv}(\tau') - \text{ftv}(\Gamma) = \{\alpha_1, \ldots, \alpha_n\} \qquad \Gamma\{x \mapsto \forall \alpha_1, \ldots, \alpha_n \,.\, \tau'\} \vdash e : \tau}{\Gamma \vdash \text{MLET}(x, e', e) : \tau}$$

(MILNER'S LET)

Here ftv is a function that finds free type variables. Operationally, we first find τ', the type of e', but we don't simply extend Γ with $\{x \mapsto \tau'\}$. Instead, using $\forall$, we *close* over all the free type variables of τ' that are not also free type variables of types in Γ. Milner's discovery was that if a type variable isn't mentioned in the environment, then you can instantiate that type variable any way you want, and you can even instantiate it *differently* at different uses of x. By closing over such type variables, we have the potential to give x a polymorphic type scheme. For example, in the following variation on `too-poly`, the let-bound variable `nil` gets a polymorphic type scheme, and when `nil` is used, the polymorphic variable is instantiated once with `int` and once with `bool`.

299 ⟨*transcript* 292⟩+≡ ◁298 307▷

```
-> (val not-too-poly
      (let ((nil '()))
         (pair (cons 1 nil) (cons #t nil))))
((1) (#t)) : int list * bool list
```

So let-bound variables might be polymorphic. Why not λ-bound variables? We can't make λ-bound variables polymorphic because we don't know what type of value a λ-bound variable might stand for—it could be any actual parameter. By contrast, we know exactly what type of value a let-bound variable stands for, because it is right there in the program; it is always the type of e'. If e' could be polymorphic (because it has type variables that don't appear in the environment), that polymorphism can be made explicit in the type scheme associated with x.

The polymorphic MLET is sometimes called "Milner's let," in honor of Robin Milner, who developed ML's type system. The fundamental new operation needed to handle Milner's let is *generalization*. We'll define a special-purpose function `generalize`, which takes as argument a type τ and a set of constrained type variables $\mathcal{A}$:

$$\texttt{generalize}(\tau, \mathcal{A}) = \forall \alpha_1, \ldots, \alpha_n \,.\, \tau, \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \text{ftv}(\tau) - \mathcal{A}.$$

Often $\mathcal{A}$ is the set of type variables that appear in a type environment, i.e., $\mathcal{A} = \text{ftv}(\Gamma)$. As suggested in Figure 7.1, generalization is a bit like the opposite of instantiation: for any τ and Γ, it's always true that $\tau <: \texttt{generalize}(\tau, \text{ftv}(\Gamma))$. An implementation of `generalize` appears in code chunk 319c.

cons ℘329b
pair ℘806c

Using `generalize`, we can rewrite the rule for Milner's let:

$$\frac{\Gamma \vdash e' : \tau' \qquad \sigma = \texttt{generalize}(\tau', \text{ftv}(\Gamma)) \qquad \Gamma\{x \mapsto \sigma\} \vdash e : \tau}{\Gamma \vdash \text{MLET}(x, e', e) : \tau}$$

(MILNER'S LET)

To extend this rule to work on a μML LET, which simultaneously binds multiple names, I add a premise for each binding "**let** $x_i = e_i$," and I add all the new type schemes $\sigma_1, \ldots, \sigma_n$ to the environment:

$$\frac{\begin{array}{c}\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \\ \sigma_i = \texttt{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \le i \le n \\ \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau\end{array}}{\Gamma \vdash \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \qquad \text{(LET)}$$

The type rule for LETREC is very similar; the difference is that the e_i's are checked in type environmment Γ', which itself contains the bindings for the x_i's. The essential idea of LETREC is that Γ' is defined using the set $\{\tau_1, \ldots, \tau_n\}$, and each τ_i is in turn defined using Γ'. So Γ' is defined in terms of itself, with types $\tau_1, \ldots, \tau_n$ as intermediaries:

$$\frac{\begin{array}{c}\Gamma' = \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \\ \Gamma' \vdash e_i : \tau_i, \quad 1 \le i \le n \\ \sigma_i = \texttt{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \le i \le n \\ \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau\end{array}}{\Gamma \vdash \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \qquad \text{(LETREC)}$$

Within Γ', the expressions e_i are *not* polymorphic. Only once all the types of the e_i's are fixed is it safe to generalize the types and compute the type of e. A LETREC is used to define a nest of mutually recursive functions, and because `generalize` is not applied until the types of all the functions are computed, these functions are *not* polymorphic when used in each other's definitions—they are polymorphic only when called from e. Even experienced ML programmers can be surprised by this restriction; functions that look polymorphic may be less polymorphic than you expected.

The rule for LETSTAR would be annoying to write down directly: there is too much bookkeeping. Instead, we treat LETSTAR as syntactic sugar, rewriting it as a nest of LETs.

$$\frac{\Gamma \vdash \text{LET}(\langle x_1, e_1\rangle, \text{LETSTAR}(\langle x_2, e_2, \ldots, x_n, e_n\rangle, e)) : \tau \qquad n > 0}{\Gamma \vdash \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \qquad \text{(LETSTAR)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{LETSTAR}(\langle\rangle, e) : \tau} \qquad \text{(EMPTYLETSTAR)}$$

Type rules for definitions

Just as in the operational semantics, a definition produces a new value environment, in the type system, a definition produces a new type environment. The relevant judgment has the form $\langle d, \Gamma\rangle \to \Gamma'$, which says that when definition d is elaborated in environment Γ, the new environment is Γ'.

A VAL binding is just like a Milner let binding, and the rules are almost the same.

$$\frac{\Gamma \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \text{ftv}(\Gamma))}{\langle \text{VAL}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}} \qquad \text{(VAL)}$$

The rule for VAL-REC requires that a recursive value be defined in the environment used to find its type.

$$\frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \text{ftv}(\Gamma))}{\langle \text{VAL-REC}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}} \qquad \text{(VALREC)}$$

A top-level expression is syntactic sugar for a binding to `it`.

$$\frac{\langle \text{VAL}(\texttt{it}, e), \Gamma\rangle \to \Gamma'}{\langle \text{EXP}(e), \Gamma\rangle \to \Gamma'} \qquad \text{(EXP)}$$

A DEFINE is syntactic sugar for a combination of VAL-REC and LAMBDA.

$$\frac{\langle \text{VAL-REC}(f, \text{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e)), \Gamma\rangle \to \Gamma'}{\langle \text{DEFINE}(f, (\langle x_1, \ldots, x_n\rangle, e)), \Gamma\rangle \to \Gamma'} \qquad \text{(DEFINE)}$$

7.5 From type rules to type inference

The type rules above don't obviously specify a deterministic algorithm for deciding if a μML term has a type. More precisely, given Γ and e, there is no obvious way to find τ such that $\Gamma \vdash e : \tau$. Difficulties arise in the LAMBDA rule, where it's not obvious what should be the types of the arguments, and in the VAR rule, where it's not obvious *which* instance of σ we should use as τ. The solution is to use fresh type variables; when we see an argument x_i, we simply record its type as α_i, where α_i is a new type variable that is not used anywhere else in the program. Think of α_i as standing for an unknown type. When we see more code, we might learn something about α_i. For example, if x_i is added to 1, then we would learn that α_i has to be equal to `int`. We would eventually *substitute* `int` for α_i.

How do we discover an appropriate type to substitute for each fresh type variable? There are two methods worth discussing:

- The first method of finding substitutions is described by simple mathematics and is easy to prove correct; it is the method of *explicit substitutions*. In this method, each individual substitution is discovered by *unification*. An implementation relies on an ML function `unify`; calling $\texttt{unify}(\tau_1, \tau_2)$ returns a θ such that $\theta(\tau_1) = \theta(\tau_2)$. The substitution θ is called a *unifier* of τ_1 and τ_2. These unifying substitutions can be composed to implement *type inference*: given Γ and e, we can find θ and τ such that $\boxed{\theta\Gamma \vdash e : \tau}$.

  The problem with the method of explicit substitutions is that type inference generates an awful lot of substitutions, and they have to be composed in exactly the right order. And there are too many wrong orders; for example, if you have three substitutions, there are six ways to compose them, and only one way is guaranteed to be correct. Even worse, it's not enough simply to compose the substitutions; you frequently have to apply them to types and to type environments. In a language as big as μML, the implementation is not so easy to get right.

- The second method of finding substitutions is described by more elaborate mathematics and is not so easy to prove correct; it is the method of *type-equality constraints*. When we want two types τ_1 and τ_2 to be equal, we *don't* unify them right away. Instead, we write down the *constraint* $\tau_1 \doteq \tau_2$, which says that τ_1 must equal τ_2. If we have two or more such constraints, we conjoin them into a single constraint using logical *and*, as in $C_1 \wedge C_2$. The constraints are used in type inference in a judgment of the form $\boxed{C, \Gamma \vdash e : \tau}$.

The great thing about the method of explicit constraints is that the composition operation $\wedge$ is associative and commutative, so the order in which you compose constraints doesn't matter. Any order is as good as any other, and as long as you conjoin all the constraints your code produces, there's no way to get the conjunction wrong. Using constraints, most of type inference is not so hard to get right.

But a price has to be paid somewhere, and to turn a judgment-with-constraints into a real judgment of the form $\Gamma \vdash e : \tau$, when we see a LET binding or a VAL binding, we have to produce a substitution that makes the constraints true. We produce the a substitution using a *constraint solver*. A constraint solver does the same job as `unify`, but a constraint solver is a bit more complicated to implement, and there's more math to understand. Once we have the constraint solver, however, implementing type inference itself is infinitely easier. For this reason, the method of constraints is the one that I recommend you implement.

Both methods of type inference are justified by the principle that we can substitute for free type variables in a valid derivation. That is, if $\dfrac{\mathcal{D}}{\Gamma \vdash e : \tau}$ is a valid derivation, then for all substitutions θ, $\dfrac{\theta\mathcal{D}}{\theta\Gamma \vdash \theta e : \theta\tau}$ is also a valid derivation.

7.5.1 The method of explicit substitutions

When the first algorithm for ML type inference was developed, it used explicit substitutions. If we want to understand explicit substitutions in detail, a good place to start is by developing a syntactic proof system for judgments of the form $\theta\Gamma \vdash e : \tau$. Here, for example, is the rule for IF.

$$\frac{\begin{array}{c}\theta_1\Gamma \vdash e_1 : \tau_1 \qquad \theta_2(\theta_1\Gamma) \vdash e_2 : \tau_2 \qquad \theta_3(\theta_2(\theta_1\Gamma)) \vdash e_3 : \tau_3 \\ \theta(\theta_3\tau_2) = \theta(\tau_3) = \tau \\ \theta'(\theta(\theta_3(\theta_2(\tau_1)))) = \theta'(\texttt{bool})\end{array}}{(\theta' \circ \theta \circ \theta_3 \circ \theta_2 \circ \theta_1)\,\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \theta'\tau} \qquad \text{(IF)}$$

This rule has an operational interpretation:

1. Infer type τ_1 for e_1, producing substitution θ_1.
2. Apply θ_1 to the environment Γ, and continue in similar fashion, inferring types and substitutions from e_2 and e_3.
3. Unify types $\theta_3\tau_2$ and τ_3, producing a new substitution θ. The resulting type τ becomes (after one more substitution) the type of the entire IF expression.
4. Unify the type of e_1 with `bool`, producing substitution θ'. Be careful to apply proper substitutions.
5. Return the composition of the substitutions, together with $\theta'\tau$.

If we take the math seriously, we have to convince ourselves that the new rule is *sound*. That means that we have to show that whenever there is a derivation using the new rule, there is a corresponding derivation using the original IF rule. Real proofs of soundness are beyond the scope of this book, but a hand-waving argument in the direction of soundness can help you understand how the whole system works. The key idea of the proof is to develop a systematic way of rewriting derivations so that if we are given a derivation in the new system, and we rewrite that derivation, we are left with a derivation in the old system. In the case of the IF rule, we can rewrite $(\theta' \circ \theta \circ \theta_3 \circ \theta_2 \circ \theta_1)\,\Gamma \to \Gamma$, $(\theta' \circ \theta \circ \theta_3 \circ \theta_2)\,\tau_1 \to \tau_1$, and so on. For example, if $\theta_1\Gamma \vdash e_1 : \tau_1$, we apply substitution $(\theta' \circ \theta \circ \theta_3 \circ \theta_2)$ to both sides, rewrite, and the premise becomes equivalent to $\Gamma \vdash e_1 : \tau_1$. A similar rewriting of all the premises enables us to apply the original IF rule to draw the conclusion. Along the way, we have to exploit a few additional facts, e.g., because `bool` is a type constructor, then for any substitution θ', $\theta'(\texttt{bool}) = \texttt{bool}$.

Although the new IF rule is sound, and it has a clear operational interpretation, the rule requires a *lot* of explicit substitutions; it's a bookkeeping nightmare. Before we abandon the method of explicit substitutions, we introduce a couple of tricks which make it possible to write the IF rule in a way that is more easily implemented.

The first trick is to extend the typing judgment to lists of expressions and types. We write $\theta\Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n$ as an abbreviation for a set of n separate judgments: $\theta\Gamma \vdash e_1 : \tau_1, \ \ldots \ , \theta\Gamma \vdash e_n : \tau_n$. When $n = 1$, this judgment degenerates to $\theta\Gamma \vdash e_1 : \tau_1$. For $n > 1$, finding the common substitution θ requires combining substitutions from different judgments.

$$\frac{\theta\Gamma \vdash e_1 : \tau_1 \qquad \theta'(\theta\Gamma) \vdash e_2, \ldots, e_n : \tau_2, \ldots, \tau_n}{(\theta' \circ \theta)\,\Gamma \vdash e_1, \ldots, e_n : \theta'\tau_1, \tau_2, \ldots, \tau_n} \qquad \text{(TypesOf)}$$

The soundness of the rule comes from applying θ' to both sides of the premise $\theta\Gamma \vdash e_1 : \tau_1$. By using this new judgment, we can reduce the number of substitutions in rules like IF.

The second trick is to reduce the number of unifications by unifying *tuple* types. For example, $\theta'(\tau_1 \times \tau_2) = \theta'(\texttt{bool} \times \tau_3)$ if and only if $\theta'(\tau_1) = \theta'(\texttt{bool})$ and $\theta'(\tau_2) = \theta'(\tau_3)$. These two tricks enable us to simplify the IF rule dramatically:

$$\frac{\theta\Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3 \qquad \theta'(\tau_1 \times \tau_2) = \theta'(\texttt{bool} \times \tau_3)}{(\theta' \circ \theta)\,\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \theta'\tau_3} \qquad \text{(If)}$$

We can prove this rule sound as before.

As another example, here is the application rule.

$$\frac{\begin{array}{c}\theta\Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \\ \theta'(\hat{\tau}) = \theta'(\tau_1 \times \cdots \times \tau_n \to \alpha), \text{ where } \alpha \text{ is fresh}\end{array}}{(\theta' \circ \theta)\,\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \theta'\alpha} \qquad \text{(Apply)}$$

The most difficult rule to express using explicit substitutions is probably LETREC. The nondeterministic rule is

$$\frac{\begin{array}{c}\Gamma' = \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \\ \Gamma' \vdash e_i : \tau_i, \quad 1 \le i \le n \\ \sigma_i = \mathtt{generalize}(\tau_i, \mathrm{ftv}(\Gamma)), \quad 1 \le i \le n \\ \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau\end{array}}{\Gamma \vdash \textsc{letrec}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \qquad \text{(LETREC)}$$

The new rule is

$$\frac{\begin{array}{c}\Gamma' = \Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}, \text{ where all } \alpha_i\text{'s are fresh} \\ \theta\Gamma' \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ \theta'\tau_i = \theta'(\theta\alpha_i), \quad 1 \le i \le n \\ \sigma_i = \mathtt{generalize}(\theta'\tau_i, \mathrm{ftv}((\theta' \circ \theta)\,\Gamma)), \quad 1 \le i \le n \\ \theta''(((\theta' \circ \theta)\,\Gamma)\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}) \vdash e : \tau\end{array}}{(\theta'' \circ \theta' \circ \theta)\,\Gamma \vdash \textsc{letrec}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \qquad \text{(LETREC)}$$

The soundness proof is a bit tricky, as it requires not only careful rewriting, but also an argument that the substitution θ'' does not change any of the type variables that are *bound* in any σ_i.

Using explicit substitutions, the IF and APPLY rules can be made relatively simple: each has a recursive call producing a θ and a call to `unify` producing a θ'. Neither substitution has to be applied to anything, and the resulting substitution is $\theta' \circ \theta$. If every rule were this simple, you'd probably be able to use explicit substitutions successfully in an implementation. But other rules, like LETREC, are far more complicated: not only must you compose three or more substitutions, but you must compose them in the right order, and you must apply certain substitutions or even compositions (like $\theta' \circ \theta$) to one or more arguments of other calls. In my experience, people who tackle this implementation task almost always forget a substitution somewhere, leading to an implementation of type inference that almost, but not quite, works. There is a better way.

7.5.2 The method of explicit constraints

To find that better way, let's return to the nondeterministic type system of Section 7.4.4. In a rule like

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau} \qquad \text{(IF)}$$

e_1 magically has the right type `bool` and e_2 and e_3 magically have the same type τ. The idea of explicit constraints is that instead of requiring magic, we allow each each e_i to have whatever type it wants, which we call τ_i. Then we insist that τ_1 must equal `bool` and τ_2 must equal τ_3. To record our insistence, we use explicit constraints, and we extend the form of typing judgments to add a constraint C to the typing context. For the IF rule, the constraint that expresses what we insist about τ_1, τ_2, and τ_3 is $\tau_1 \doteq \mathtt{bool} \wedge \tau_2 \doteq \tau_3$. The $\doteq$ and $\wedge$ notations are explained below.

A judgment of the form $C, \Gamma \vdash e : \tau$ means "assuming the constraint C is satisfied, in environment Γ term e has type τ." Constraints are formed by conjoining *simple equality constraints*:

- A simple equality constraint has the form $\tau_1 \doteq \tau_2$ (pronounced "τ_1 must equal τ_2"), and it is satisfied if and only if the types τ_1 and τ_2 are equal.

- A conjunction has the form $C_1 \wedge C_2$ (pronounced "C_1 and C_2"), and it is satisfied only if both C_1 and C_2 are satisfied. Just as in ordinary logic, conjunction of constraints is associative and commutative.

The math and the code become easier if we allow a third form of constraint:

- The *trivial constraint* has the form $\mathbf{T}$, and it is always considered satisfied. The trivial constraint is a left and right identity of $\wedge$. The constraint may be pronounced "trivial" or "true."

Formally, here is a grammar for constraints:

$$C ::= \tau_1 \doteq \tau_2 \mid C_1 \wedge C_2 \mid \mathbf{T}$$

In a typing judgment, the constraint captures what has to be true if a term is going to have a type. A typing judgment has the form $C, \Gamma \vdash e : \tau$. Operationally, the expression e and environment Γ are the inputs to the type checker; the type τ and constraint C are the outputs. The type system is designed so that if $C, \Gamma \vdash e : \tau$ is derivable, and if constraint C is satisfied, then erasing all the constraints produces a derivation of $\Gamma \vdash e : \tau$ that is valid in the original, nondeterministic type system.

Using constraints, we can write the IF rule properly. We must not only create new constraints $\tau_1 \doteq \texttt{bool}$ and $\tau_2 \doteq \tau_3$; we must also remember any old constraints used to give types to the subexpressions e_1, e_2, and e_3. "Old" constraints propagate from the premises of a rule to the conclusion.

$$\frac{C_1, \Gamma \vdash e_1 : \tau_1 \qquad C_2, \Gamma \vdash e_2 : \tau_2 \qquad C_3, \Gamma \vdash e_3 : \tau_3}{C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \doteq \texttt{bool} \wedge \tau_2 \doteq \tau_3, \Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau_2} \quad (\textsc{If})$$

The conclusion of this rule shows the conjunction of three old constraints with two new simple equality constraints. For the IF expression to have a type, all the constraints needed to give types to e_1, e_2, and e_3 must be satisfied. And it must also be the case that $\tau_1 \doteq \texttt{bool}$ and $\tau_2 \doteq \tau_3$. If all the constraints are satisfied, which implies that $\tau_1 = \texttt{bool}$ and $\tau_2 = \tau_3$ (actual equality, with no dots), then the rule is equivalent to the original IF rule.

Constraints require a lot less bookkeeping than do the substitutions in Section 7.5.1, but it's still worth defining a typing judgment that describes lists of expressions and types. We write $C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n$ as an abbreviation for a set of n separate judgments, where C is the conjunction of the constraints of the individual judgments:

$$\frac{C_1, \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad C_n, \Gamma \vdash e_n : \tau_n}{C_1 \wedge \cdots \wedge C_n, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n} \quad (\textsc{TypesOf})$$

Using this notation, we can make rules a little simpler. For example, here is the simplified IF rule:

$$\frac{C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3}{C \wedge \tau_1 \doteq \texttt{bool} \wedge \tau_2 \doteq \tau_3, \Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau_2} \quad (\textsc{If})$$

As another example, here is the application rule. The type $\hat{\tau}$ of the thing applied must be a function type, and furthermore, it must be a function type whose arguments are types $\tau_1, \ldots, \tau_n$. But we don't know the return type. Here, as we do whenever we don't know a type, we make the return type a fresh variable α, and we let its ultimate identity be determined by the constraint C. The type $\hat{\tau}$ of function e must therefore be equal to $\tau_1 \times \cdots \times \tau_n \to \alpha$:

$$\frac{C, \Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \doteq \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \alpha} \qquad \text{(APPLY)}$$

Again, if the constraints are satisfied, the rule is equivalent to the original.

With these rules, we can do an example. Let's assume that the type environment Γ contains these bindings:

$$\Gamma = \{+ : \forall.\texttt{int} \times \texttt{int} \to \texttt{int}, \texttt{cons} : \forall\alpha.\alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list}\}.$$

This Γ has no free type variables. Using Γ, if we try to infer a type for `(+ 1 2)`, the derivation looks roughly like this, where α_{10} is a fresh type variable:

$$\frac{\dfrac{\dfrac{}{\mathbf{T}, \Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int} \doteq \texttt{int}} \quad \dfrac{}{\mathbf{T}, \Gamma \vdash 1 : \texttt{int}} \quad \dfrac{}{\mathbf{T}, \Gamma \vdash 2 : \texttt{int}}}{\mathbf{T} \wedge \mathbf{T} \wedge \mathbf{T} \wedge \texttt{int} \times \texttt{int} \to \texttt{int} \doteq \texttt{int} \times \texttt{int} \to \alpha_{10}, \Gamma \vdash \texttt{(+ 1 2)} : \alpha_{10}}}{\cdots}$$

If we then substitute `int` for α_{10} in this derivation, we get

$$\frac{\dfrac{\dfrac{}{\mathbf{T}, \Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int} \doteq \texttt{int}} \quad \dfrac{}{\mathbf{T}, \Gamma \vdash 1 : \texttt{int}} \quad \dfrac{}{\mathbf{T}, \Gamma \vdash 2 : \texttt{int}}}{\mathbf{T} \wedge \mathbf{T} \wedge \mathbf{T} \wedge \texttt{int} \times \texttt{int} \to \texttt{int} \doteq \texttt{int} \times \texttt{int} \to \texttt{int}, \Gamma \vdash \texttt{(+ 1 2)} : \texttt{int}}}{\cdots}$$

All the constraints are satisfied, and if we erase them, we are left with a derivation in the original, nondeterministic system.

Converting nondeterministic rules to use constraints

Almost every nondeterministic rule can be converted to a deterministic, constraint-based rule. We need just two ideas:

- If in the original rule the same type τ appears in more than one place, give each appearance its own name (like τ_2 and τ_3), and introduce constraints forcing the names to be equal.
- If a type τ appears in the original rule and we don't know what τ is supposed to be, use a fresh type variable.

The first idea is illustrated by the IF rule. To illustrate the second idea, let's convert the nondeterministic VAR rule to use explicit constraints. The nondeterministic rule says

$$\frac{\Gamma(x) = \sigma \qquad \tau' <: \sigma}{\Gamma \vdash x : \tau'} \tag{VAR}$$

We know that $\tau' <: \sigma$ when $\sigma = \forall\alpha_1, \ldots, \alpha_n.\tau$, and some (unknown) types are substituted for the $\alpha_1, \ldots, \alpha_n$. For the unknown types, we pick fresh type variables $\alpha'_1, \ldots, \alpha'_n$.

$$\frac{\begin{array}{c}\Gamma(x) = \forall\alpha_1, \ldots, \alpha_n.\tau \\ \alpha'_1, \ldots, \alpha'_n \text{ are fresh and distinct}\end{array}}{\mathbf{T}, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \cdots \circ (\alpha_n \mapsto \alpha'_n))\,\tau} \tag{VAR}$$

In τ, each α_i is replaced by the corresponding fresh α'_i, and $\alpha'_1, \ldots, \alpha'_n$ are eventually constrained by the way x is used. For example, if we infer a type for the expression `(cons 1 '())`, although the instance of `cons` uses a fresh type variable, eventually that type variable should be constrained to be equal to `int`. The derivation looks roughly like this:

$$\frac{\dfrac{\Gamma(\texttt{cons}) = \forall\alpha.\alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list}}{\mathbf{T}, \Gamma \vdash \texttt{cons} : \alpha_{11} \times \alpha_{11}\ \texttt{list} \to \alpha_{11}\ \texttt{list}} \qquad \frac{}{\mathbf{T}, \Gamma \vdash \texttt{1} : \texttt{int}} \qquad \frac{}{\mathbf{T}, \Gamma \vdash \texttt{'()} : \alpha_{12}\ \texttt{list}}}{\mathbf{T} \wedge \mathbf{T} \wedge \alpha_{11} \times \alpha_{11}\ \texttt{list} \to \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list} \to \alpha_{13}, \Gamma \vdash \texttt{(cons 1 '())} : \alpha_{13}}$$

We can take that big constraint and rewrite it as a simpler, equivalent constraint:

$$\alpha_{11} \doteq \texttt{int} \wedge \alpha_{11}\ \texttt{list} \doteq \alpha_{12}\ \texttt{list} \wedge \alpha_{11}\ \texttt{list} \doteq \alpha_{13}.$$

Both constraints are solved by the substitution

$$(\alpha_{11} \mapsto \texttt{int}) \circ (\alpha_{12} \mapsto \texttt{int}) \circ (\alpha_{13} \mapsto \texttt{int list}),$$

and the type of `(cons 1 '())` is `int list`.

Inferring polytypes for bound names

In the examples above, each type variable is determined to be a completely known type, like `int` or `int list`. But when we define polymorphic values, some types will be unknown. These types will be denoted by free type variables, and eventually we want to create polymorphic type schemes that use $\forall$ with those free type variables. For example, a function that makes a singleton list should have a polymorphic type:

307 ⟨*transcript* 292⟩+≡ ◁299 310▷

```
-> (val singleton (lambda (x) (cons x '())))
singleton : forall 'a . 'a -> 'a list
```

cons 𝒫329b

Converting from a monotype τ with free type variables to a type scheme σ with an explicit `forall` is the most challenging part of type inference; this conversion is called *generalization*.

Our first example of generalization is the easiest: it's the rule for a top-level VAL binding:

$$\frac{\begin{array}{c} C, \Gamma \vdash e : \tau \\ \theta C \text{ is satisfied} \qquad \theta\Gamma = \Gamma \\ \sigma = \texttt{generalize}(\theta\tau, \text{ftv}(\Gamma)) \end{array}}{\langle \text{VAL}(x, e), \Gamma \rangle \to \Gamma\{x \mapsto \sigma\}} \qquad \text{(VAL)}$$

Here's an operational interpretation:

- Typecheck e in environment Γ, getting back type τ and constraint C
- Choose a substitution θ such that θC is satisfied, making sure that θ does not affect any free type variables of Γ (this step is called *solving* C; how to do it is the subject of Section 7.5.3 below)
- Generalize the type $\theta\tau$ to form the general type scheme σ, which becomes the type of x in a new environment $\Gamma\{x \mapsto \sigma\}$

Assuming we have a valid derivation of the judgment $C, \Gamma \vdash e : \tau$, here's why the rule works: for any θ,

- $\theta C, \theta\Gamma \vdash e : \theta\tau$ is a derivable judgment.
- Because $\theta\Gamma = \Gamma$, the judgment $\theta C, \Gamma \vdash e : \theta\tau$ is also derivable.
- Because θC is satisfied, $\Gamma \vdash e : \theta\tau$ is a derivable judgment in the original, nondeterministic system.
- If the original system can derive type $\theta\tau$ for e, it is safe to generalize that type.

The VAL rule illustrates the key ideas underlying constraint-based inference of polymorphic type schemes:

- We need a substitution θ that solves a constraint C, but θ mustn't substitute for any free type variables of the environment Γ.
- Using substitution θ on a type τ gives us a type we can generalize.

Here's an example. If we elaborate the definition

```
(val singleton (lambda (x) (cons x '())))
```

then the the body of the `lambda` is checked in the extended environment $\Gamma' = \Gamma\{\mathtt{x} : \forall.\alpha_{14}\}$, and the derivation of the type of the `lambda` is going to look something like this:[5]

$$\dfrac{\dfrac{\dfrac{\Gamma'(\mathtt{cons}) = \forall\alpha.\alpha \times \alpha\ \mathtt{list} \to \alpha\ \mathtt{list}}{\mathbf{T}, \Gamma \vdash \mathtt{cons} : \alpha_{15} \times \alpha_{15}\ \mathtt{list} \to \alpha_{15}\ \mathtt{list}} \quad \dfrac{\Gamma'(\mathtt{x}) = \forall.\alpha_{14}}{\mathbf{T}, \Gamma' \vdash \mathtt{x} : \alpha_{14}} \quad \dfrac{}{\mathbf{T}, \Gamma \vdash \mathtt{'()} : \alpha_{16}\ \mathtt{list}}}{\alpha_{15} \times \alpha_{15}\ \mathtt{list} \to \alpha_{15}\ \mathtt{list} \doteq \alpha_{14} \times \alpha_{16}\ \mathtt{list} \to \alpha_{17}, \Gamma' \vdash \mathtt{(cons\ x\ '())} : \alpha_{17}}}{\alpha_{15} \times \alpha_{15}\ \mathtt{list} \to \alpha_{15}\ \mathtt{list} \doteq \alpha_{14} \times \alpha_{16}\ \mathtt{list} \to \alpha_{17}, \Gamma \vdash \mathtt{(lambda\ (x)\ (cons\ x\ '()))} : \alpha_{14} \to \alpha_{17}}$$

The constraint can be simplified to

$$\alpha_{15} \doteq \alpha_{14} \wedge \alpha_{15}\ \mathtt{list} \doteq \alpha_{16}\ \mathtt{list} \wedge \alpha_{15}\ \mathtt{list} \doteq \alpha_{17}$$

and then further simplified to

$$C = \alpha_{15} \doteq \alpha_{14} \wedge \alpha_{15} \doteq \alpha_{16} \wedge \alpha_{15}\ \mathtt{list} \doteq \alpha_{17}.$$

This constraint is solved by several substitutions; for example, it is solved by the substitution $\theta = (\alpha_{14} \mapsto \alpha_{15}) \circ (\alpha_{16} \mapsto \alpha_{15}) \circ (\alpha_{17} \mapsto \alpha_{15}\ \mathtt{list})$. The most interesting premises of the VAL are then

$$C, \Gamma \vdash \mathtt{(lambda\ (x)\ (cons\ x\ '()))} : \alpha_{17}$$

$$\tau = \alpha_{14} \to \alpha_{17} \qquad \theta\tau = \alpha_{15} \to \alpha_{15}\ \mathtt{list} \qquad \sigma = \forall\alpha_{15}.\alpha_{15} \to \alpha_{15}\ \mathtt{list}$$

The name `singleton` is therefore added to Γ with type scheme $\forall\alpha_{15}.\alpha_{15} \to \alpha_{15}\ \mathtt{list}$, which we prefer to write in canonical form as $\forall\alpha.\alpha \to \alpha\ \mathtt{list}$.

A term that has no type produces an unsolvable constraint

In the method of explicit substitutions, when a term has no type, the problem manifests as a call to `unify` with two types that can't be unified. In the method of explicit constraints, when a term has no type, the problem manifests as a constraint that can't be solved. As an example, let's consider the function

```
(lambda (x) (cons x x))
```

Let's assume that `x` is introduced to the environment with the monotype $\forall.\alpha_{18}$, that `cons` is instantiated with type $\alpha_{19} \times \alpha_{19}\ \mathtt{list} \to \alpha_{19}\ \mathtt{list}$, and that the return type of the `lambda` is type variable α_{20}. Then the system derives a judgment that looks roughly like this:

$$\alpha_{19} \times \alpha_{19}\ \mathtt{list} \to \alpha_{19}\ \mathtt{list} \doteq \alpha_{18} \times \alpha_{18} \to \alpha_{20}, \Gamma \vdash \mathtt{(lambda\ (x)\ (cons\ x\ x))} : \alpha_{18} \to \alpha_{20}.$$

The constraint can be simplified to

$$\alpha_{19} \doteq \alpha_{18} \wedge \alpha_{19}\ \mathtt{list} \doteq \alpha_{18} \wedge \alpha_{20} \doteq \alpha_{20}.$$

[5] Just as in the nondeterministic system, the rule for `lambda` type-checks the body in an extended environment that binds the formal parameter `x` to a monotype. If you wonder why I don't show you a rule for `lambda`, it's because I want you to develop the rule yourself; see Exercise 4 on page 334.

The third simple type equality is already satisfied, and the first can be satisfied by substituting α_{19} for α_{18} or vice versa. So this constraint is solvable if and only if the simpler constraint

$$\alpha_{19}\ \texttt{list} \doteq \alpha_{19}$$

is solvable (or equivalently, if $\alpha_{18}\ \texttt{list} \doteq \alpha_{18}$ is solvable). But there is no possible substitution for α_{19} that makes α_{19} `list` equal to α_{19}.[6] And after putting the unsolvable constraint into canonical form, that's what the interpreter reports:

310 ⟨*transcript* 292⟩+≡ ◁307 313▷

```
-> (val broken (lambda (x) (cons x x)))
type error: cannot make 'a equal to 'a list
```

Adding constraints is OK

We're about to bump up against one of the tradeoffs I made in the design of the languages in this book. I want you to be able to write interesting programs, so μML, like μScheme (and like full Scheme and full ML), allows you to write `let` bindings that have multiple names in them. But the ability to write interesting programs has to be paid for, and here's the cost: a type system that allows multiple names in a `let` binding is significantly more complicated than one that has only a single-name `let` binding. Most authors want to avoid this cost, which is why most presentations of programming-language theory work with a simplified "core calculus." But if you're going to build an interpreter that infers types, you don't want some simplified core calculus; you want to infer types for code you've actually written. So in the next subsection, we'll tackle generalization and Milner's `let` using the real μML `let` constructs. But first, I'm going to slip in an extra rule:

$$\frac{C_i, \Gamma \vdash e_i : \tau_i \qquad C \text{ has a solution}}{C \wedge C_i, \Gamma \vdash e_i : \tau_i} \qquad \text{(OVERCONSTRAINED)}$$

The idea is that although the constraint C_i is *sufficient* to give e_i a type, we can add another constraint C, provided that C has a solution. To prove this rule sound, we appeal to the substitution θ that solves C: if we apply θ to the valid derivation associated with the judgment $C_i, \Gamma \vdash e_i : \tau_i$, we get another valid derivation. With this new rule in our pockets, we can tackle the full μML `let`.

[6] For a proof, consider the number of times that `list` appears on each side. No matter what you substitute for α_{19}, there will always be one more `list` constructor on the left than on the right, so the two sides can never be equal.

Generalization in Milner's let binding

The most difficult part of constraint-based type inference is understanding how to generalize at a LET binding. Here are the elements of a solution:

- Given an expression e, we make a normal recursive call which produces a type τ and a constraint C.
- We solve the constraint C, then keep, as a new constraint C', that part of the solution which applies to type variables that are mentioned in the environment.
- We refine type τ by applying the solution to C.
- We generalize the refined type only over variables that are mentioned neither in the environment nor in the restricted constraint.

To show how these elements are combined, I cheat a little bit: for a few moments, I pretend that an expression can be given a type *scheme*, not just a type. This bit of pretend enables us to focus just on generalization without worrying about binding. Don't try it at home.

$$\frac{\begin{array}{c} C, \Gamma \vdash e : \tau \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \doteq \theta\alpha \mid \alpha \in \operatorname{dom}\theta \cap \operatorname{ftv}(\Gamma)\} \\ \sigma = \texttt{generalize}(\theta\tau, \operatorname{ftv}(\Gamma) \cup \operatorname{ftv}(C')) \end{array}}{C', \Gamma \vdash e : \sigma} \qquad \text{(IDEA OF GENERALIZATION)}$$

The notation $\bigwedge\{\ldots\}$ is new; it says to form a single constraint by conjoining the elements of the set. If the set is empty, the constraint formed is $\mathbf{T}$.

Here's an operational interpretation:

- Typecheck e in environment Γ, getting back type τ and constraint C
- Solve C, getting solution θ.
- Apply the solution to τ.
- Restrict the solution to type variables that appear free in Γ, and use that restriction to form a new constraint

$$C' = \bigwedge\{\alpha \doteq \theta\alpha \mid \alpha \in \operatorname{dom}\theta \cap \operatorname{ftv}(\Gamma)\}.$$

Why does this rule work? Even an informal explanation is pretty elaborate, so I've put it in a sidebar.

Sidebar: Generalization with constraints

Here is an informal argument about how generalization works with constraints.

- To compute the type being generalized, we have a judgment $C, \Gamma \vdash e : \tau$. For example, we might have

$$\alpha \doteq \texttt{int}, \Gamma \vdash \texttt{(+ x 1)} : \alpha,$$

  where $\Gamma = \{\texttt{x} : \alpha\}$. Even though the type of `(+ x 1)` has a type variable, we can't possibly generalize that variable: α *has* to be `int`.

- If we solve the constraint C, we get the substitution $\theta = (\alpha \mapsto \texttt{int})$. The type τ is an output, so we can substitute `bool` for α there, getting $\theta\tau = \texttt{int}$. But Γ is an input, so there's no way to substitute there: the constraint has to be retained.

- In the general case, what we are doing is taking the constraint C and its solution θ, and we are splitting θ into two parts, so that $\theta = \theta_g \circ \theta_l$, where
  - $\theta_l = \theta|_{\operatorname{dom}\theta \setminus \operatorname{ftv}(\Gamma)}$ is the *local* part; it substitutes for type variables that are mentioned in C (and possibly in τ) but never in Γ. Substitution θ_l can be applied to τ and then thrown away.
  - $\theta_g = \theta|_{\operatorname{ftv}(\Gamma)}$ is the *global* part; it substitutes for type variables that are mentioned in Γ. It can be applied to τ but not to Γ. We can't through θ_g away, so we convert it to constraint C'.

- Because θ is idempotent and none of the variables in $\operatorname{dom}\theta$ is generalized, $\theta\sigma = \sigma$.

If we have a valid derivation of $\theta C', \theta\Gamma \vdash e : \theta\sigma$, we can use the equalities above to recover a valid derivation in the original, nondeterministic system.

To get from the idea of generalization to a rule that applies to μML's LET, we use generalized type schemes σ_i to extend the environment, then typecheck the body in the extended environment, producing type τ under constraint C_b. The whole LET expression has type τ and the conjoined constraints $C' \wedge C_b$:

$$\frac{\begin{array}{c} C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \doteq \theta\alpha \mid \alpha \in \operatorname{dom}\theta \cap \operatorname{ftv}(\Gamma)\} \\ \sigma_i = \texttt{generalize}(\theta\tau_i, \operatorname{ftv}(\Gamma) \cup \operatorname{ftv}(C')), \quad 1 \le i \le n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \quad (\textsc{Let})$$

The rule for LETREC is a little fancier: the e_i's are checked in an extended environment where each x_i is bound to a fresh type variable α_i, and *before* generalization, each type τ_i is constrained to be equal to the corresponding α_i. The rest is the same.

$$\frac{\begin{array}{c}\Gamma' = \Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\},\ \text{where all } \alpha_i\text{'s are distinct and fresh} \\ C_r, \Gamma' \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ C = C_r \wedge \tau_1 \doteq \alpha_1 \wedge \cdots \wedge \tau_n \doteq \alpha_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \doteq \theta\alpha \mid \alpha \in \operatorname{dom}\theta \cap \operatorname{ftv}(\Gamma)\} \\ \sigma_i = \texttt{generalize}(\theta\tau_i, \operatorname{ftv}(\Gamma) \cup \operatorname{ftv}(C')), \quad 1 \le i \le n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau\end{array}}{C' \wedge C_b, \Gamma \vdash \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \quad \text{(LETREC)}$$

Here's an example of generalization in LET:

313 ⟨*transcript* 292⟩+≡ ◁310

```
-> (val ss (lambda (y)
             (let ((single (lambda (x) (cons x '()))))
               (single (single y)))))
ss : forall 'a . 'a -> 'a list list
```

The whole derivation is too big to show, so let's look at pieces. We start by assuming that the *body* of the outer `lambda` is type-checked in an environment

$$\Gamma = \{\texttt{cons} : \forall\alpha.\alpha \times \alpha\ \texttt{list} \to \alpha, \texttt{y} : \alpha_{20}\}.$$

In a similar environment, an example above shows a derivation for the inner `lambda`, so if we write $C = \alpha_{22} \times \alpha_{22}\ \texttt{list} \to \alpha_{22}\ \texttt{list} \doteq \alpha_{21} \times \alpha_{23}\ \texttt{list} \to \alpha_{24}$, we can take for granted that

$$C, \Gamma \vdash \texttt{(lambda (x) (cons x '()))} : \alpha_{21} \to \alpha_{24}$$

is derivable. Now the LET rule comes into play:

$$\begin{aligned} C &= \alpha_{22} \times \alpha_{22}\ \texttt{list} \to \alpha_{22}\ \texttt{list} \doteq \alpha_{21} \times \alpha_{23}\ \texttt{list} \to \alpha_{24} \\ \tau_1 &= \alpha_{21} \to \alpha_{24} \\ \theta &= (\alpha_{21} \mapsto \alpha_{22}) \circ (\alpha_{23} \mapsto \alpha_{22}) \circ (\alpha_{24} \mapsto \alpha_{22}\ \texttt{list}) \\ C' &= \textstyle\bigwedge\{\,\} = \mathbf{T} \\ \operatorname{ftv}(\Gamma) &= \{\alpha_{20}\} \\ \sigma_1 &= \texttt{generalize}(\alpha_{22} \to \alpha_{22}\ \texttt{list}, \operatorname{ftv}(\Gamma)) = \forall\alpha_{22}.\alpha_{22} \to \alpha_{22}\ \texttt{list} \end{aligned}$$

cons 𝒫 329b

The body of the LET, `(single (single x))`, is checked in the extended environment $\Gamma_e = \Gamma\{\texttt{single} : \forall\alpha_{22}.\alpha_{22} \to \alpha_{22}\ \texttt{list}\}$. Each instance of `single` gets its own type:

$$\dfrac{\dfrac{\dfrac{\Gamma_e(\texttt{single}) = \forall\alpha_{22}.\alpha_{22} \to \alpha_{22}\ \texttt{list}}{\mathbf{T}, \Gamma_e \vdash \texttt{single} : \alpha_{25} \to \alpha_{25}\ \texttt{list}} \quad \dfrac{\Gamma_e(\texttt{y}) = \forall.\alpha_{20}}{\mathbf{T}, \Gamma_e \vdash \texttt{y} : \alpha_{20}}}{\alpha_{25} \to \alpha_{25}\ \texttt{list} \doteq \alpha_{20} \to \alpha_{26}, \Gamma_e \vdash \texttt{(single y)} : \alpha_{26}} \quad \dfrac{\Gamma_e(\texttt{single}) = \forall\alpha_{22}.\alpha_{22} \to \alpha_{22}\ \texttt{list}}{\mathbf{T}, \Gamma \vdash \texttt{single} : \alpha_{27} \to \alpha_{27}\ \texttt{list}}}{\alpha_{25} \to \alpha_{25}\ \texttt{list} \doteq \alpha_{20} \to \alpha_{26} \wedge \alpha_{27} \to \alpha_{27}\ \texttt{list} \doteq \alpha_{26} \to \alpha_{28}, \Gamma \vdash \texttt{(single (single y))} : \alpha_{28}}$$

The type of the outer `lambda` is therefore $\alpha_{20} \to \alpha_{28}$, with constraint $C' \wedge C_b$, which is

$$\mathbf{T} \wedge \alpha_{25} \to \alpha_{25}\ \texttt{list} \doteq \alpha_{20} \to \alpha_{26} \wedge \alpha_{27} \to \alpha_{27}\ \texttt{list} \doteq \alpha_{26} \to \alpha_{28}.$$

This constraint is equivalent to

$$\alpha_{25} \doteq \alpha_{20} \wedge \alpha_{25}\ \texttt{list} \doteq \alpha_{26} \wedge \alpha_{27} \doteq \alpha_{26} \wedge \alpha_{27}\ \texttt{list} \doteq \alpha_{28},$$

which is solved by the substitution

$$(\alpha_{25} \mapsto \alpha_{20}) \circ (\alpha_{26} \mapsto \alpha_{20}\ \texttt{list}) \circ (\alpha_{27} \mapsto \alpha_{20}\ \texttt{list}) \circ (\alpha_{28} \mapsto \alpha_{20}\ \texttt{list list}).$$

The type of the outer `lambda` is therefore $\alpha_{20} \to \alpha_{20}\ \texttt{list list}$, and at the VAL binding, this type is generalized to the type scheme $\forall\alpha_{20}.\alpha_{20} \to \alpha_{20}\ \texttt{list list}$.

7.5.3 Solving constraints

As shown by the examples above, the method of explicit constraints reduces the type-inference problem to a constraint-solving problem. A fresh type variable that is introduced to represent an unknown type may get constrained, and by solving the constraints, we learn what (if anything) to substitute for each type variable. Substitutions play a crucial role, but they are used only to infer types at LET and VAL, where potentially polymorphic names are bound. In this section we explore all the details. I hope you'll use the ideas to implement your own constraint solver.

We begin with a little proof system that defines "satisfied" by induction over the structure of constraints:

$$\dfrac{\tau_1 = \tau_2}{\tau_1 \doteq \tau_2 \text{ is satisfied}} \qquad \dfrac{C_1 \text{ is satisfied} \qquad C_2 \text{ is satisfied}}{C_1 \wedge C_2 \text{ is satisfied}} \qquad \dfrac{}{\mathbf{T} \text{ is satisfied}}$$

Constraints, like types, can be converted to other constraints by substitution: when we apply a substitution to a constraint, we may replace type variables in that constraint. We *solve* a constraint by finding a substitution that, when applied to the constraint, produces a constraint that is satisfied. A substitution is applied to a constraint using the following rules:

$$\theta(\tau_1 \doteq \tau_2) = \theta\tau_1 \doteq \theta\tau_2 \qquad \theta(C_1 \wedge C_2) = \theta C_1 \wedge \theta C_2 \qquad \theta\mathbf{T} = \mathbf{T}$$

You can see our ML representation of constraints in chunk 320b and the substitution function in chunk 320e.

When θC is satisfied, we say that θ *solves* C. Not all constraints can be solved; for example, there is no substitution that solves `int` $\doteq$ `bool`, and there is no substitution that solves $\alpha \doteq \alpha$ `list`.

As Exercise 6 on page 334 asks you to show, satisfaction is a property that is preserved by substitution. Formally, if θC is satisfied, and if θ' is another substitution, then $\theta'(\theta C)$ is also satisfied. Another way of writing this fact is that $(\theta' \circ \theta)C$ is satisfied.

At bottom, constraint solving is the same problem as unification. For example, if you have a constraint solver, then any substitution that solves the constraint $\tau_1 \doteq \tau_2$ is a unifier of τ_1 and τ_2. And if you have a unifier, you can use it to solve constraints—the exact trick is the subject of Exercise 13 on page 336.

Cases for constraint solving

A constraint solver is given a constraint C and returns a substitution θ such that θC is satisfied. The solver need consider only three cases:

- C is the trivial constraint $\mathbf{T}$, in which case θ_I solves C
- C is the conjunction $C_1 \wedge C_2$, in which case both sub-constraints C_1 and C_2 must be solved
- C is a simple equality constraint of the form $\tau_1 \doteq \tau_2$

Both conjunctions and simple equality constraints require attention to detail.

Wouldn't it be great if you could solve a conjunction $C_1 \wedge C_2$ simply by solving C_1, then solving C_2, then composing the solutions? It's a pity you can't. Here's a useless rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \qquad \tilde{\theta}_2 C_2 \text{ is satisfied}}{(\tilde{\theta}_2 \circ \theta_1)(C_1 \wedge C_2) \text{ may or may not be satisfied}} \qquad \text{(UnsolvedConjunction)}$$

Even when θ_1 solves C_1 and $\tilde{\theta}_2$ solves C_2, neither $\tilde{\theta}_2 \circ \theta_1$ nor $\theta_1 \circ \tilde{\theta}_2$ is guaranteed to solve $C_1 \wedge C_2$. I've put a tilde on the substitution $\tilde{\theta}_2$ because I'm going to treat it as the bad guy: looking at $\tilde{\theta}_2 \circ \theta_1$, here's a quick explanation of what goes wrong:

- We want $(\tilde{\theta}_2 \circ \theta_1)(C_1 \wedge C_2)$ to be satisfied. According the rule for satisfying conjunctions shown above, this means we need both $(\tilde{\theta}_2 \circ \theta_1)C_1$ and $(\tilde{\theta}_2 \circ \theta_1)C_2$ to be satisfied.
- By the assumption that θ_1 solves C_1, $\theta_1 C_1$ is satisfied. And because satisfaction is preserved by substitution, $(\tilde{\theta}_2 \circ \theta_1)C_1$ is satisfied. So that part is good.
- We also need $(\tilde{\theta}_2 \circ \theta_1)C_2 = \tilde{\theta}_2(\theta_1 C_2)$ to be satisfied. But unfortunately, all we know about $\tilde{\theta}_2$ is that it solves C_2, and this information isn't enough to guarantee that it also solves $\theta_1 C_2$.

You can carry this line of thinking a little further by completing Exercises 7 and 8 on page 335, which ask you for a proof and some examples.

The thing that goes wrong with the simple, obvious composition of substitutions is that in the last step the constraint we need solved is not C_2; it is $\theta_1 C_2$. This observation leads to a technique that works, which I express here as a useful inference rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \qquad \theta_2(\theta_1 C_2) \text{ is satisfied}}{(\theta_2 \circ \theta_1)(C_1 \wedge C_2) \text{ is satisfied}} \qquad \text{(SolvedConjunction)}$$

Substitution θ_2, which may be different from $\tilde{\theta}_2$, does what we need. The rule has an operational interpretation involving a recursive call to the solver, the application of substitution θ_1, another recursive call to the solver, and finally the composition of two substitutions. Exercise 9 on page 335 ask you to prove that this rule is sound. As a corollary, the operational interpretation works as a solver.

Informally, the way I like to think about solving conjunction is that if θ_1 solves C_1, then θ_1 represents the *assumptions* that I have to make if C_1 is to have a solution. If those assumptions are to hold everywhere, then they somehow must be accounted for when I look at C_2. And the way to account for assumptions is to *apply* the corresponding substitution.

Solving simple equality constraints

The last remaining case for a constraint solver is a simple equality constraint of the form $\tau_1 \doteq \tau_2$. I call this case "simple," but the code is far from simple: because there are three ways to form a type, there are *nine* possible ways to form a simple equality constraint. Nine cases is a lot to code, but if you organize your code carefully, you can use the power of ML pattern matching to cut them down. And no matter what case you are considering, the goal is always them same: find a θ such that $\theta\tau_1 = \theta\tau_2$. Let's tackle the most tricky case first, the easy cases next, and the most involved case last.

The tricky case is one in which the left-hand side is a type variable, giving the constraint the form $\alpha \doteq \tau_2$. You might hope that this constraint is always solved by the substitution $(\alpha \mapsto \tau_2)$. You might be disappointed:

- If τ_2 does not mention α, then $(\alpha \mapsto \tau_2)\tau_2 = \tau_2$, and also $(\alpha \mapsto \tau_2)\alpha = \tau_2$. We win!
- If τ_2 is *equal* to α, then $(\alpha \mapsto \alpha)$ is the identity substitution θ_I. We win again.
- If τ_2 *mentions* α but is *not equal* to α, then it turns out that the constraint $\alpha \doteq \tau_2$ has no solution. (The proof is the subject of Exercise 10.)

Type τ_2 mentions α if and only if α *occurs free in* τ_2. The part of your code that checks to see if this happens is called the *occurs check*.

Suppose the left-hand side of the constraint is not a type variable, but the right-hand side is. That is, we have a constraint of the form $\tau_1 \doteq \alpha$. Well, this constraint has exactly the same solutions as $\alpha \doteq \tau_1$, and you can just swap the two sides and make a recursive call to your solver.

If neither side is a type variable, then what we are left with is combinations of type constructors and type applications (`TYCON` and `CONAPP`). Because every substitution maps constructors to constructors and applications to applications, we needn't think very hard about constraints that equate the two: they can't be solved. We also don't have to think hard if each side of the constraint holds a type constructor: every substitution leaves every constructor unchanged, so a constraint of the form $\mu \doteq \mu$ is solved by the identity substitution, and a constraint of the form $\mu \doteq \mu'$, where $\mu \neq \mu'$, has no solution.

The most complicated case is a constraint in which both sides are constructor applications. Because every substitution must preserve the structure of a constructor application (see the substitution laws on page 293), we can break down such a constraint into a conjunction of smaller constraints:

$$\frac{\theta(\tau \doteq \tau' \wedge \tau_1 \doteq \tau_1' \wedge \cdots \wedge \tau_n \doteq \tau_n') \text{ is satisfied}}{\theta(\textsc{conapp}(\tau, \langle \tau_1, \ldots, \tau_n \rangle) \doteq \textsc{conapp}(\tau', \langle \tau_1', \ldots, \tau_n' \rangle)) \text{ is satisfied}} \quad (\textsc{SolveConappConapp})$$

Operationally, the solver is presented with the constraint on the bottom. It uses the elements of this constraint to construct the constraint on the top, on which it then calls itself recursively. Exercise 11 on page 335 asks you to prove that this rule is sound.

An example of constraint solving

Using the ideas above, let's solve a constraint. The constraint

$$\mathbf{T} \wedge (\mathbf{T} \wedge \alpha_{11} \times \alpha_{11}\ \texttt{list} \to \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list} \to \alpha_{13}) \tag{7.1}$$

is big enough to be interesting, but a little too big for a complete, formal derivation. So let's solve it informally.

The constraint is a conjunction, so the first task is to solve the left conjunct, which is $\mathbf{T}$. This conjunct is solved by the identity substitution θ_I, which we then apply to the right conjunct

$$\mathbf{T} \wedge \alpha_{11} \times \alpha_{11}\ \texttt{list} \to \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list} \to \alpha_{13}. \tag{7.2}$$

The identity substitution leaves this conjunct unchanged, and we continue solving recursively. The exact same set of steps lead us to solve

$$\alpha_{11} \times \alpha_{11}\ \texttt{list} \to \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list} \to \alpha_{13}. \tag{7.3}$$

At this point the final case for a simple equality constraint comes into play: on both sides, we have an application of the $\to$ constructor. We use the SOLVECONAPPCONAPP rule to convert this constraint to

$$(\to \doteq \to) \wedge (\alpha_{11} \times \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list} \wedge \alpha_{11}\ \texttt{list} \doteq \alpha_{13}). \tag{7.4}$$

The left conjunct, $(\to \doteq \to)$, has two equal type constructors and so is solved by θ_I which, when applied to the right conjunct, leaves it unchanged. So we solve

$$\alpha_{11} \times \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list} \wedge \alpha_{11}\ \texttt{list} \doteq \alpha_{13}. \tag{7.5}$$

The next step is to solve the left conjunct

$$\alpha_{11} \times \alpha_{11}\ \texttt{list} \doteq \texttt{int} \times \alpha_{12}\ \texttt{list}, \tag{7.6}$$

which requires another application of the SOLVECONAPPCONAPP rule, asking us to solve

$$(\times \doteq \times) \wedge \alpha_{11} \doteq \texttt{int} \wedge \alpha_{11}\ \texttt{list} \doteq \alpha_{12}\ \texttt{list}. \tag{7.7}$$

We must next solve

$$\alpha_{11} \doteq \texttt{int} \wedge \alpha_{11}\ \texttt{list} \doteq \alpha_{12}\ \texttt{list}, \tag{7.8}$$

and we begin with its left conjunct

$$\alpha_{11} \doteq \texttt{int}. \tag{7.9}$$

Finally we have a case with a type variable on the left, and constraint 7.9 is solved by $\theta_1 = \alpha_{11} \mapsto \texttt{int}$. We then apply θ_1 to the constraint $\alpha_{11}\ \texttt{list} \doteq \alpha_{12}\ \texttt{list}$, yielding

$$\texttt{int list} \doteq \alpha_{12}\ \texttt{list}. \tag{7.10}$$

Let's not go through all the steps; constraint 7.10 is solved by $\theta_2 = \alpha_{12} \mapsto \texttt{int}$. Constraints 7.6, 7.7, and 7.8 are therefore solved by the composition $\theta_2 \circ \theta_1 = (\alpha_{12} \mapsto \texttt{int} \circ \alpha_{11} \mapsto \texttt{int})$.

Now we can return to constraint 7.5. We apply $\theta_2 \circ \theta_1$ to the right conjunct $\alpha_{11}\ \texttt{list} \doteq \alpha_{13}$, yielding

$$\texttt{int list} \doteq \alpha_{13}, \tag{7.11}$$

which is solved by substitution $\theta_3 = \alpha_{13} \mapsto \texttt{int list}$. The solution to constraint 7.5 is therefore $\theta_3 \circ \theta_2 \circ \theta_1$, which is

$$\theta = \theta_3 \circ \theta_2 \circ \theta_3 = \alpha_{13} \mapsto \texttt{int list} \circ \alpha_{12} \mapsto \texttt{int} \circ \alpha_{11} \mapsto \texttt{int}.$$

Substitution θ also solves constraints 7.1 to 7.4.

7.6 The interpreter

Most of our interpreter for μML is just like Chapter 5's interpreter for μScheme. We have added type inference. Significant parts of the implementation of type inference don't appear here, however, because they are left as Exercises.

7.6.1 Functions on types and type schemes

This section defines several families of functions that are used throughout type inference.

Sets of type variables

Much of type inference manipulates sets of type variables. We provide a simple implementation that represents a set using a list with no duplicate elements.[7]

318a ⟨*sets of free type variables* 318a⟩≡ (294c) 318b▷

```
type 'a set = 'a list
val emptyset = []
fun member x =
  List.exists (fn y => y = x)
fun insert (x, ys) =
  if member x ys then ys else x::ys
fun union (xs, ys) = foldl insert ys xs
fun inter (xs, ys) =
  List.filter (fn x => member x ys) xs
fun diff  (xs, ys) =
  List.filter (fn x => not (member x ys)) xs
```

```
type 'a set
emptyset : 'a set
member   : ''a -> ''a set -> bool
insert   : ''a     * ''a set  -> ''a set
union    : ''a set * ''a set  -> ''a set
inter    : ''a set * ''a set  -> ''a set
diff     : ''a set * ''a set  -> ''a set
```

The function `freetyvars` computes the free type variables of a type. For readability, I ensure that type variables appear in the set in the order of their first appearance in the type, when reading from left to right.

318b ⟨*sets of free type variables* 318a⟩+≡ (294c) ◁318a

```
fun freetyvars t =
  let fun f (TYVAR v,          ftvs) = insert (v, ftvs)
        | f (TYCON _,          ftvs) = ftvs
        | f (CONAPP (ty, tys), ftvs) = foldl f (f (ty, ftvs)) tys
  in  rev (f (t, emptyset))
  end
```

```
freetyvars : ty -> name set
```

Fresh type variables

CONAPP 291
TYCON 291
TYVAR 291

A type variable that does not appear in any type environment or substitution is called *fresh*. When inferring the type of a function, we need fresh type variables to stand for the (unknown) types of the arguments. When instantiating a polytype, we need fresh type variables to stand for the (unknown) types at which the polytype is instantiated. A fresh type variable is also useful to stand for the return type in a function application.

[7] The ML types of the set operations include type variables with double primes, like ''a. The type variable ''a can be instantiated only with an "equality type." Equality types include base types like strings and integers, as well as user-defined types that do not contain functions. Functions *cannot* be compared for equality.

In our interpreter, fresh type variables come from the `freshtyvar` function. We use a private mutable counter to supply an arbitrary number of type variables of the form `t`n. Because μML does not permit users to write type variables explicitly, we need not worry about the names colliding with other names.

319a ⟨*Hindley-Milner types* 291⟩+≡ ◁296b 319b▷

```
local
  val n = ref 1
in
  fun freshtyvar _ = TYVAR ("t" ^ Int.toString (!n) before n := !n + 1)
end
```

freshtyvar : 'a -> ty

Canonical type schemes

Type variables like `'t136` are not very readable in error messages. A type scheme like `forall 't136 . 't136 list -> int` is unpleasant to look at, and it is completely interchangeable with the equivalent type scheme `forall 'a . 'a list -> int`. The two schemes are equivalent because if a type variable is ∀-bound, its name is irrelevant. For readability, we are better off using the names `'a`, `'b`, etc., for bound type variables. The function `canonicalize` renames bound type variables in a type scheme. We replace each existing bound type variable with a canonically named type variable, being careful not to use the name of any free type variable.

319b ⟨*Hindley-Milner types* 291⟩+≡ ◁319a 319c▷

```
fun canonicalize (FORALL (bound, ty)) =
  let fun canonicalTyvarName n =
        if n < 26 then str (chr (ord #"a" + n))
        else "v" ^ Int.toString (n - 25)
      val free = diff (freetyvars ty, bound)
      fun unusedIndex n =
        if member (canonicalTyvarName n) free then unusedIndex (n+1) else n
      fun newBoundVars (index, [])                = []
        | newBoundVars (index, oldvar :: oldvars) =
            let val n = unusedIndex index
            in  canonicalTyvarName n :: newBoundVars (n+1, oldvars)
            end
      val newBound = newBoundVars (0, bound)
  in  FORALL (newBound, tysubst (bindList (bound, map TYVAR newBound, emptyEnv)) ty)
  end
```

canonicalize : type_scheme -> type_scheme
newBoundVars : int * name list -> name list

bindList	214
diff	318a
emptyEnv	214
FORALL	291
freetyvars	318b
member	318a
tysubst	294b
TYVAR	291

The function `unusedIndex` finds a name for a bound type variable; it ensures that the name is not the name of any free type variable.

Generalization and instantiation

Calling `generalize`$(\tau, \mathcal{A})$ generalizes type τ to a type scheme by closing over type variables not in $\mathcal{A}$. It also puts the type scheme into canonical form.

319c ⟨*Hindley-Milner types* 291⟩+≡ ◁319b 320a▷

```
fun generalize (tau, tyvars) =
  canonicalize (FORALL (diff (freetyvars tau, tyvars), tau))
```

generalize : ty * name set -> type_scheme

The dual function, `instantiate`, is defined in chunk 294d. It requires a list of types with which to instantiate, but the common case is to instantiate with fresh type variables. Function `freshInstance` implements this case.

320a ⟨*Hindley-Milner types* 291⟩+≡ ◁319c 320b▷

```
fun freshInstance (FORALL (bound, tau)) =
  instantiate (FORALL (bound, tau), map freshtyvar bound)
```

`freshInstance : type_scheme -> ty`

Constraints

To highlight the relationship between the code and the math, I've chosen a representation that's close to the math:[8] the $\doteq$ operator is `=*=`; the $\wedge$ operator is `/\`; and the $\mathbf{T}$ constraint is `TRIVIAL`.

320b ⟨*Hindley-Milner types* 291⟩+≡ ◁320a 320c▷

```
datatype con = =*= of ty  * ty
             | /\  of con * con
             | TRIVIAL
infix 4 =*=
infix 3 /\
```

Appendix H defines a function `constraintString`, which we use to print constraints. It also defines a function `untriviate`, which removes all trivial conjuncts from a constraint.

320c ⟨*Hindley-Milner types* 291⟩+≡ ◁320b 320d▷

⟨*printing constraints* (generated automatically)⟩

`constraintString : con -> string`

Now that we can represent constraints, we can find free type variables in a constraint, and we can substitute for free type variables.

320d ⟨*Hindley-Milner types* 291⟩+≡ ◁320c 320e▷

```
fun freetyvarsConstraint (t =*= t') = union (freetyvars t, freetyvars t')
  | freetyvarsConstraint (c /\  c') = union (freetyvarsConstraint c,
                                            freetyvarsConstraint c')
  | freetyvarsConstraint TRIVIAL    = emptyset
```

A substitution is applied to a constraint using the following rules:

$$\theta(\tau_1 \doteq \tau_2) = \theta\tau_1 \doteq \theta\tau_2 \qquad \theta(C_1 \wedge C_2) = \theta C_1 \wedge \theta C_2 \qquad \theta\mathbf{T} = \mathbf{T}$$

The code is quite similar to the code for `tysubst` in chunk 294b.

320e ⟨*Hindley-Milner types* 291⟩+≡ ◁320d 320f▷

```
fun consubst theta =
  let fun subst (tau1 =*= tau2) = tysubst theta tau1 =*= tysubst theta tau2
        | subst (c1 /\ c2)      = subst c1 /\ subst c2
        | subst TRIVIAL         = TRIVIAL
  in  subst
  end
```

`consubst : subst -> con -> con`

We implement the $\bigwedge\{\cdots\}$ operator using the ML function `conjoinConstraints`. I avoid using `foldr` or `foldr` because I want to preserve the number and order of sub-constraints.

320f ⟨*Hindley-Milner types* 291⟩+≡ ◁320e 321a▷

```
fun conjoinConstraints []     = TRIVIAL
  | conjoinConstraints [c]    = c
  | conjoinConstraints (c::cs) = c /\ conjoinConstraints cs
```

`conjoinConstraints : con list -> con`

emptyset 318a
FORALL 291
freetyvars 318b
freshtyvar 319a
instantiate 294d
type ty 291
tysubst 294b
union 318a

[8] Experienced type-system hackers might prefer a list of pairs of types, but a list of pairs is easy to work with only if you already understand what's going on.

7.6.2 Constraint solving

Our type-inference algorithm is built on a constraint solver, which given a constraint C produces a substitution θ such that θC is trivially satisfied. But if the algorithm is given an ill-typed program, it produces an *unsolvable* constraint: one for which no such θ exists. Examples of unsolvable constraints include `int` $\doteq$ `bool` and α `list` $\doteq \alpha$. When we discover an unsolvable constraint, we want to issue a readable error message, which shouldn't be full of machine-generated fresh type variables. To do so, we should take the pair of types that can't be made equal, and we should put the *pair* into canonical form. Function `unsatisfiableEquality` does just that, by putting the two types together into a pair type.

321a ⟨*Hindley-Milner types* 291⟩+≡ ◁320f 321c▷

```
exception TypeError of string

fun unsatisfiableEquality (t1, t2) =
  case canonicalize (FORALL (union (freetyvars t1, freetyvars t2), tupletype [t1, t2]))
    of FORALL (_, CONAPP (TYCON "tuple", [t1, t2])) =>
         raise TypeError ("cannot make " ^ typeString t1 ^ " equal to " ^ typeString t2)
     | _ => let exception ThisCan'tHappen in raise ThisCan'tHappen end
```

I'm hoping you will implement the solver.

321b ⟨*constraint solving* [[**prototype**]] 321b⟩≡

```
exception LeftAsExercise of string
fun solve c = raise LeftAsExercise "solve"
```

```
solve : con -> subst
```

321c ⟨*Hindley-Milner types* 291⟩+≡ ◁321a 322a▷

⟨*constraint solving* (generated automatically)⟩

For debugging, it can be useful to check to see if a substitution actually solves a constraint.

321d ⟨*constraint standardization and splitting* 321d⟩≡

```
fun isSolved TRIVIAL = true
  | isSolved (tau =*= tau') = eqType (tau, tau')
  | isSolved (c /\ c') = isSolved c andalso isSolved c'
fun solves (theta, c) = isSolved (consubst theta c)
```

```
isSolved : con -> bool
solves : subst * con -> bool
```

7.6.3 Type environments

Whenever we call `generalize`, we use the free type variables of some type environment. And a type environment contains the type of every name ever defined in a μML program, so it can get big. We therefore pay a little bit of attention to efficiency. In particular, instead of searching an entire type environment every time we want free type variables, we pay a little extra when we *extend* a type environment—and we arrange to find free type variables in constant time.

Here are the operations we perform on type environments:

- Function `bindtyscheme` adds a binding $x : \sigma$ to the environment Γ. We use `bindtyscheme` to implement the LAMBDA rule and the various LET rules.
- Function `findtyscheme` looks up a variable x to find σ such that $\Gamma(x) = \sigma$. We use `findtyscheme` to implement the VAR rule.

canonicalize 319b
CONAPP 291
FORALL 291
freetyvars 318b
tupletype 296b
TYCON 291
typeString 693
union 318a

- Function `freetyvarsGamma` finds the type variables free in Γ, i.e., the type variables free in any σ in Γ. We need `freetyvarsGamma` to get a set of free type variables to use in `generalize`; when we assign a type scheme to a let-bound variable, only those type variables not free in Γ may be $\forall$-bound.

If `freetyvarsGamma` were implemented over a simple representation of type `type_scheme env`, it would visit every type scheme in every binding in the environment. Because most bindings contribute no free type variables, most visits would be unnecessary. We therefore make an optimization: with every type environment, we keep a cache of the free type variables. Our representation of type environments is therefore as follows:

322a ⟨*Hindley-Milner types* 291⟩+≡ ◁321c 322b▷

```
type type_env = type_scheme env * name set
```

We create an empty type environment that binds no variables and has no free type variables. And when we try to find a type scheme, we ignore the free variables.

322b ⟨*Hindley-Milner types* 291⟩+≡ ◁322a 322c▷

```
val emptyTypeEnv =
      (emptyEnv, emptyset)
fun findtyscheme (v, (Gamma, free)) = find (v, Gamma)
```

```
emptyTypeEnv : type_env
findtyscheme : name * type_env -> type_scheme
```

When we add a new binding, we update the set of free type variables in Γ. We take the union of the existing free type variables with the free type variables of the new type scheme σ.

322c ⟨*Hindley-Milner types* 291⟩+≡ ◁322b 322d▷

```
fun bindtyscheme (v, sigma as FORALL (bound, tau), (Gamma, free)) =
  (bind (v, sigma, Gamma), union (diff (freetyvars tau, bound), free))
```

```
bindtyscheme : name * type_scheme * type_env -> type_env
```

Finally, when we want the free type variables, we just take them from the pair.

322d ⟨*Hindley-Milner types* 291⟩+≡ ◁322c

```
fun freetyvarsGamma (_, free) = free
```

```
freetyvarsGamma : type_env -> name set
```


7.6.4 Type inference

Type inference for expressions

Given an expression e and type environment Γ, `typeof(e, Γ)` returns a pair (τ, C) such that $C, \Gamma \vdash e : \tau$. It uses three auxiliary functions.

⟨*type inference* 322e⟩≡ (331c) 324d▷

```
fun typeof (e, Gamma) =
  let ⟨function typesof, to infer the types of a list of expressions 323a⟩
      ⟨function literal, to infer the type of a literal constant (left as an exercise)⟩
      ⟨function ty, to infer the type of an expression, given Gamma 323b⟩
  in  ty e
  end
```

```
typeof  : exp      * type_env -> ty      * con
typesof : exp list * type_env -> ty list * con
literal : value -> ty * con
ty      : exp   -> ty * con
```

Sidebar: Why generalize and instantiate?

We use quantified types (i.e., type schemes) so we can instantiate them when we look them up in an environment. The reason we have to instantiate is to get the full effect of polymorphism. If we didn't, we wouldn't be able to type terms such as `(1::nil, true::nil)`. Suppose we had an environment Γ with only types, not type schemes:

$$\Gamma = \{1 : \texttt{int}, \mathit{true} : \texttt{bool}, \mathit{nil} : \alpha\ \texttt{list}, :: : \alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list}\}$$

Then when type-checking `1::nil` we would get the constraint $\alpha \doteq \texttt{int}$. And when type-checking `1::nil` we would get the constraint $\alpha \doteq \texttt{bool}$. But the conjoined constraint $\alpha \doteq \texttt{int} \wedge \alpha \doteq \texttt{bool}$ has no solution, and type checking would fail.
Instead, we use `freshInstance` to make sure that every use of a polymorphic value (here `::` and `nil`) has a type different from any other instance. In order to make that work, the environment has to contain polytypes:

$$\Gamma = \{1 : \forall.\texttt{int}, \mathit{true} : \forall.\texttt{bool}, \mathit{nil} : \forall\alpha\,.\,\alpha\ \texttt{list}, :: : \forall\alpha\,.\,\alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list}\}$$

Now, we can imagine our sample term like this, writing `::` as a prefix operator so as to show the types:

```
(op :: : 't121 * 't121 list -> 't121 list (1    : int,  nil : 't122 list),
 op :: : 't123 * 't123 list -> 't123 list (true : bool, nil : 't124 list))
```

The constraint $\texttt{'t121} \doteq \texttt{int} \wedge \texttt{int} \doteq \texttt{'t122} \wedge \texttt{'t123} \doteq \texttt{bool} \wedge \texttt{bool} \doteq \texttt{'t124}$ does have a solution, and the whole term has the type `int list * bool list`, as desired.

Calling `typesof`$(\langle e_1, \ldots, e_n\rangle, \Gamma)$ returns $(\langle \tau_1, \ldots, \tau_n\rangle, C)$ such that $C, \Gamma \vdash e_i : \tau_i$ for every i with $1 \le i \le n$. The base case is trivial; the induction step uses this rule from Section 7.5.2:

$$\frac{C_1, \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad C_n, \Gamma \vdash e_n : \tau_n}{C_1 \wedge \cdots \wedge C_n, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n} \quad (\textsc{TypesOf})$$

323a ⟨*function* `typesof`, *to infer the types of a list of expressions* 323a⟩≡ (322e)

```
fun typesof ([],    Gamma) = ([], TRIVIAL)
  | typesof (e::es, Gamma) =
      let val (tau,  c)  = typeof  (e,  Gamma)
          val (taus, c') = typesof (es, Gamma)
      in  (tau :: taus, c /\ c')
      end
```

findtyscheme 322b
freshInstance 320a
Gamma 322e
LITERAL 287a
literal 804b
TRIVIAL 320b
typeof 322e
VAR 287a

To infer the type of a literal value, we call `literal`. To infer the type of a variable, we use fresh type variables to create a most general instance of the variable's type scheme in Γ. No constraint is needed.

323b ⟨*function* `ty`, *to infer the type of an expression, given* `Gamma` 323b⟩≡ (322e)

```
fun ty (LITERAL n) = literal n
  | ty (VAR x) = (freshInstance (findtyscheme (x, Gamma)), TRIVIAL)
  ⟨more alternatives for ty 324a⟩
```

Computing the type of a literal value is left as part of Exercise 15.

323c ⟨*function* `literal`, *to infer the type of a literal constant* **[[prototype]]** 323c⟩≡

```
fun literal _ = raise LeftAsExercise "literal"
```

Section 7.5.2 shows how to rewrite a type rule to introduce explicit substitutions; here is the rule for application:

$$\frac{C, \Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \doteq \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \alpha} \qquad (\textsc{Apply})$$

To implement this rule, we let `funty` stand for $\hat{\tau}$, `actualtypes` stand for $\tau_1, \ldots, \tau_n$, and `rettype` stand for α. The first premise is implemented by a call to `typesof` and the second by a call to `freshtyvar`. The constraint is represented just as written in the rule.

324a ⟨*more alternatives for* ty 324a⟩≡ (323b) 324b▷

```
| ty (APPLY (f, actuals)) =
    (case typesof (f :: actuals, Gamma)
       of ([], _) => let exception ThisCan'tHappen in raise ThisCan'tHappen end
        | (funty :: actualtypes, c) =>
            let val rettype = freshtyvar ()
            in  (rettype, c /\ (funty =*= funtype (actualtypes, rettype)))
            end)
```

The remaining cases for `ty` are left as exercises, except we provide syntactic sugar for `LETSTAR`.

324b ⟨*more alternatives for* ty 324a⟩+≡ (323b) ◁324a

```
| ty (LETX (LETSTAR, [], body)) = ty body
| ty (LETX (LETSTAR, (b :: bs), body)) =
    ty (LETX (LET, [b], LETX (LETSTAR, bs, body)))
```

324c ⟨*more alternatives for* ty **[[prototype]]** 324c⟩≡

```
| ty (IFX (e1, e2, e3))        = raise LeftAsExercise "type for IFX"
| ty (BEGIN es)                 = raise LeftAsExercise "type for BEGIN"
| ty (LAMBDA (formals, body))   = raise LeftAsExercise "type for LAMBDA"
| ty (LETX (LET, bs, body))     = raise LeftAsExercise "type for LET"
| ty (LETX (LETREC, bs, body)) = raise LeftAsExercise "type for LETREC"
```


Elaboration and type inference for definitions

Given a definition, we want to extend the top-level type environment. We infer the type of the thing defined, generalize it, and and add a binding to the environment. This step is called *elaboration*. To report to a user, we also return a string suitable for printing.

⟨*type inference* 322e⟩+≡ (331c) ◁322e

```
fun elabdef (d, Gamma) =
  case d
    of VAL    (x, e)       => ⟨infer and bind type for VAL    (x, e) 325a⟩
     | VALREC (x, e)       => ⟨infer and bind type for VALREC (x, e) 325b⟩
     | EXP e               => elabdef (VAL ("it", e), Gamma)
     | DEFINE (x, lambda) => elabdef (VALREC (x, LAMBDA lambda), Gamma)
     | USE filename        => raise RuntimeError
                                "internal error -- 'use' reached elabdef"
```

`elabdef : def * type_env -> type_env * string`

`EXP` and `DEFINE` are syntactic sugar.

The cases for `VAL` and `VALREC` resemble each other. We begin with `VAL`, which computes a type and generalizes it.

$$\frac{\begin{array}{c}C, \Gamma \vdash e : \tau \\ \theta C \text{ is satisfied} \qquad \theta\Gamma = \Gamma \\ \sigma = \mathtt{generalize}(\theta\tau, \mathrm{ftv}(\Gamma))\end{array}}{\langle \textsc{val}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}} \qquad (\textsc{Val})$$

325a ⟨*infer and bind type for* `VAL (x, e)` 325a⟩≡ (324d)

```
let val (tau, c) = typeof (e, Gamma)
    val theta    = solve c
    val sigma    = generalize (tysubst theta tau, freetyvarsGamma Gamma)
in  (bindtyscheme (x, sigma, Gamma), typeSchemeString sigma)
end
```

This code takes a big shortcut: we just assume that $\theta\Gamma = \Gamma$. How can we get away with this assumption? Because we can prove that *a top-level* Γ *never contains a free type variable.* This property guarantees that $\theta\Gamma = \Gamma$ for any θ. You can prove this property for yourself in Exercise 5 on page 334.

The code for `VALREC` is a bit more complicated. We need an environment that binds x to τ, but we don't yet know τ. The original rule looks like this:

$$\frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau \qquad \sigma = \mathtt{generalize}(\tau, \mathrm{ftv}(\Gamma))}{\langle \textsc{val-rec}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}} \qquad (\textsc{ValRec})$$

Here's a version with constraints:

$$\frac{\begin{array}{c}C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \qquad \alpha \text{ is fresh} \\ \theta(C \wedge \alpha \doteq \tau) \text{ is satisfied} \qquad \theta\Gamma = \Gamma \\ \sigma = \mathtt{generalize}(\theta\alpha, \mathrm{ftv}(\Gamma))\end{array}}{\langle \textsc{val-rec}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}} \qquad (\textsc{ValRec with constraints})$$

As usual, we introduce a fresh type variable to stand for τ, then constrain it to be equal to the type of e.

325b ⟨*infer and bind type for* `VALREC (x, e)` 325b⟩≡ (324d)

```
let val alpha    = freshtyvar ()
    val Gamma'   = bindtyscheme (x, FORALL ([], alpha), Gamma)
    val (tau, c) = typeof (e, Gamma')
    val theta    = solve (c /\ alpha =*= tau)
    val sigma    = generalize (tysubst theta alpha, freetyvarsGamma Gamma)
in  (bindtyscheme (x, sigma, Gamma), typeSchemeString sigma)
end
```


=*= 320b
bindtyscheme 322c
FORALL 291
freetyvarsGamma 322d
freshtyvar 319a
Gamma 324d
generalize 319c
solve 804a
typeof 322e
typeSchemeString 295c
tysubst 294b

7.6.5 Evaluation

Evaluation of expressions

Because the abstract syntax of μML is a subset of μScheme, the evaluator is almost a subset of the μScheme evaluator. One difference is that because μML doesn't have mutation, environments map names to values, instead of mapping them to mutable cells. Another is that type inference should eliminate most potential errors. If one of those errors occurs anyway, we raise the exception `BugInTypeInference`.

326a ⟨*evaluation* 326a⟩≡ (330a) 327b▷

```
eval : exp * value env -> value
```

```
⟨definition of separate (generated automatically)⟩
fun eval (e, rho) =
  let fun ev (LITERAL n)         = n
        | ev (VAR x)             = find (x, rho)
        | ev (IFX (e1, e2, e3)) = ev (if bool (ev e1) then e2 else e3)
        | ev (LAMBDA l)          = CLOSURE (l, fn _ => rho)
        | ev (BEGIN es) =
            let fun b (e::es, lastval) = b (es, ev e)
                  | b (   [], lastval) = lastval
            in  b (es, BOOL false)
            end
        | ev (APPLY (f, args)) =
           (case ev f
              of PRIMITIVE prim => prim (map ev args)
               | CLOSURE clo => ⟨apply closure clo to args 326b⟩
               | _ => raise BugInTypeInference "Applied non-function"
               )
        ⟨more alternatives for ev 326c⟩
  in  ev e
  end
```

To apply a closure, we bind formal parameters directly to the values of actual parameters, not to mutable cells.

⟨*apply closure* clo *to* args 326b⟩≡ (326a)

```
let val ((formals, body), mkRho) = clo
    val actuals = map ev args
in  eval (body, bindList (formals, actuals, mkRho ()))
    handle BindListLength =>
        raise BugInTypeInference "Wrong number of arguments to closure"
end
```

LET evaluates all right-hand sides in ρ, then extends ρ to evaluate the body.

⟨*more alternatives for* ev 326c⟩≡ (326a) 326d▷

```
| ev (LETX (LET, bs, body)) =
    let val (names, values) = ListPair.unzip bs
    in  eval (body, bindList (names, map ev values, rho))
    end
```

LETSTAR evaluates pairs in sequence, adding a binding to ρ after each evaluation.

326d ⟨*more alternatives for* ev 326c⟩+≡ (326a) ◁326c 327a▷

```
| ev (LETX (LETSTAR, bs, body)) =
    let fun step ((n, e), rho) = bind (n, eval (e, rho), rho)
    in  eval (body, foldl step rho bs)
    end
```

LETREC is the most interesting case. Function `makeRho'` builds an environment in which each right-hand side stands for a closure. Each closure's captured environment is the one built by `makeRho'`. The recursion is OK because the environment is built lazily, so `makeRho'` always terminates. The right-hand sides must be lambda abstractions.

327a ⟨*more alternatives for* ev 326c⟩+≡ (326a) ◁326d

```
| ev (LETX (LETREC, bs, body)) =
    let fun makeRho' () =
          let fun step ((n, e), rho) =
                    (case e
                       of LAMBDA l => bind (n, CLOSURE (l, makeRho'), rho)
                        | _ => raise RuntimeError "non-lambda in letrec")
          in  foldl step rho bs
          end
    in  eval (body, makeRho'())
    end
```

Evaluating definitions

Evaluating a definition can produce a new environment. The function `evaldef` also returns a string which, if nonempty, should be printed to show the value of the item.

327b ⟨*evaluation* 326a⟩+≡ (330a) ◁326a 328a▷

```
fun evaldef (d, rho) =
  case d
    of VAL    (name, e)      =>
          let val v   = eval (e, rho)
              val rho = bind (name, v, rho)
          in  (rho, showVal name v)
          end
     | VALREC (name, LAMBDA lambda) =>
          let fun makeRho' () = bind (name, CLOSURE (lambda, makeRho'), rho)
              val v           = CLOSURE (lambda, makeRho')
          in  (makeRho'(), showVal name v)
          end
     | VALREC _ => raise RuntimeError "expression in val-rec must be lambda"
     | EXP e    =>
          let val v   = eval (e, rho)
              val rho = bind ("it", v, rho)
          in  (rho, valueString v)
          end
     | DEFINE (name, lambda) => evaldef (VALREC (name, LAMBDA lambda), rho)
     | USE filename => raise RuntimeError "internal error -- 'use' reached evaldef"
and showVal name v =
      case v
        of CLOSURE   _ => name
         | PRIMITIVE _ => name
         | _ => valueString v
```

`evaldef : def * value env -> value env * string`

The implementation of `VALREC` works only for `LAMBDA` expressions because these are the only expressions for which we can compute the value without having the environment.

7.6.6 Primitives

Here are the primitives. All are either binary or unary operators. Type inference should guarantee that operators are used with the correct arity.

328a ⟨*evaluation* 326a⟩+≡ (330a) ◁327b 328b▷

```
unaryOp  : (value         -> value) -> (value list -> value)
binaryOp : (value * value -> value) -> (value list -> value)
fun binaryOp f = (fn [a, b] => f (a, b) | _ => raise BugInTypeInference "arity 2")
fun unaryOp  f = (fn [a]    => f  a     | _ => raise BugInTypeInference "arity 1")
```

Arithmetic primitives expect and return integers. Each primitive operation must be associated with a type scheme in the initial environment. It is easier, however, to associate a *type* with each primitive and to generalize them all at one go when we create the initial environment.

328b ⟨*evaluation* 326a⟩+≡ (330a) ◁328a 328d▷

```
arithOp   : (int * int -> int) -> (value list -> value)
arithtype : ty
fun arithOp f =
      binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
                 | _ => raise BugInTypeInference "arithmetic on non-numbers")
val arithtype = funtype ([inttype, inttype], inttype)
```

As before, we use the chunk ⟨*primops* :: 328c⟩ to put all the primitives into one giant list, and we use that list to build the initial environment for the read-eval-print loop. The big difference is that in μML, each primitive has a type as well as a value.

328c ⟨*primops* :: 328c⟩≡ (331a) 329a▷

```
("+", arithOp op +,   arithtype) ::
("-", arithOp op -,   arithtype) ::
("*", arithOp op *,   arithtype) ::
("/", arithOp op div, arithtype) ::
```

We have two kinds of predicates: ordinary predicates take one argument, and comparisons take two. Some comparisons apply only to integers. We get to reuse `embedPredicate` for the definitions.

328d ⟨*evaluation* 326a⟩+≡ (330a) ◁328b

```
predOp     : (value         -> bool) -> (value list -> value)
comparison : (value * value -> bool) -> (value list -> value)
intcompare : (int   * int   -> bool) -> (value list -> value)
predtype   : ty -> ty
comptype   : ty -> ty
fun predOp f     = unaryOp  (embedPredicate f)
fun comparison f = binaryOp (embedPredicate f)
fun intcompare f =
      comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                   | _ => raise BugInTypeInference "comparing non-numbers")
fun predtype x = funtype ([x],    booltype)
fun comptype x = funtype ([x, x], booltype)
```

booltype 296b
BugInType-Inference 294d
embedPredicate 216b
funtype 296b
inttype 296b
NUM 287c

The predicates are similar to μScheme predicates. As in μScheme, values of any type can be compared for equality. Equality has type $\alpha \times \alpha \to$ `bool`, which gets generalized to type scheme $\forall\alpha\,.\,\alpha \times \alpha \to$ `bool`. In full ML, values of function types may not be compared for equality.

329a ⟨*primops* :: 328c⟩+≡ (331a) ◁328c 329b▷

```
("<", intcompare op <, comptype inttype) ::
("=", intcompare op >, comptype inttype) ::
("=", comparison (fn (NIL,     NIL)     => true
                   | (NUM n1,  NUM n2)  => n1 = n2
                   | (SYM v1,  SYM v2)  => v1 = v2
                   | (BOOL b1, BOOL b2) => b1 = b2
                   | _                  => false)
     , comptype alpha) ::
("null?", predOp (fn NIL => true | _ => false), predtype (listtype alpha)) ::
```

The list primitives are easy:

329b ⟨*primops* :: 328c⟩+≡ (331a) ◁329a 329c▷

```
("cons", binaryOp (fn (a, b) => PAIR (a, b)),
                  funtype ([alpha, listtype alpha], listtype alpha)) ::
("car",  unaryOp  (fn (PAIR (car, _)) => car
                    | NIL => raise RuntimeError "car applied to empty list"
                    | _   => raise BugInTypeInference "car applied to non-list"),
                  funtype ([listtype alpha], alpha)) ::
("cdr",  unaryOp  (fn (PAIR (_, cdr)) => cdr
                    | NIL => raise RuntimeError "cdr applied to empty list"
                    | _   => raise BugInTypeInference "cdr applied to non-list"),
                  funtype ([listtype alpha], listtype alpha)) ::
```

The last two primitives are `print` and `error`.

329c ⟨*primops* :: 328c⟩+≡ (331a) ◁329b

```
("print", unaryOp (fn x => (print (valueString x ^ "\n"); x)),
          funtype ([alpha], unittype)) ::
("error", unaryOp (fn x => raise RuntimeError (valueString x)),
          funtype ([alpha], beta)) ::
```

The type of `error`, $\forall\alpha,\beta\,.\,\alpha \to \beta$, tells us something interesting about its behavior. The type suggests that `error` can produce an arbitrary β without ever consuming one. Such a miracle is impossible; what the type tells us is that the `error` function never returns normally. In μML this type means it must either halt the interpreter or fail to terminate; in full ML, a function of this type could also raise an exception.

7.6.7 Processing definitions: elaboration and evaluation

As in Typed Impcore and Typed μScheme, we process a definition by first elaborating it (which includes inferring its type), then evaluating it. The elaborator and evaluator produce strings that respectively represent type and value. If the value string is nonempty, we print both strings. If `e` is not well typed, calling `typeof(e, Gamma)` raises the `TypeError` exception, and we never call `eval`.

330a ⟨*checking and evaluation* 330a⟩≡ (331c) 330b▷

```
elabEvalDef : def * env_bundle * (string->unit) -> env_bundle
⟨evaluation 326a⟩
type env_bundle = type_env * value env
fun elabEvalDef (d, envs as (Gamma, rho), echo) =
  case d
    of USE filename => use readCheckEvalPrint mlSyntax filename envs
     | _ =>
       let val (Gamma, tystring)  = elabdef (d, Gamma)
           val (rho,   valstring) = evaldef (d, rho)
           val _ = if size valstring > 0 then echo (valstring ^ " : " ^ tystring)
                   else ()
       in  (Gamma, rho)
       end
```

As in Typed μScheme, `elabEvalDef` preserves the phase distinction: type inference is independent of `rho` and `evaldef`.

7.6.8 The read-eval-print loop

The read-eval-print loop is almost identical to the read-eval-print loop for Typed μScheme; the only difference is that instead of `BugInTypeChecking`, we have `BugInTypeInference`.

330b ⟨*checking and evaluation* 330a⟩+≡ (331c) ◁330a

```
readCheckEvalPrint : def stream * (string->unit) * (string->unit) -> env_bundle -> env_bundle
and readCheckEvalPrint (defs, echo, errmsg) envs =
  let fun processDef (def, envs) =
            let fun continue msg = (errmsg msg; envs)
            in  elabEvalDef (def, envs, echo)
                handle IO.Io {name, ...} => continue ("I/O error: " ^ name)
                ⟨more read-eval-print handlers 330c⟩
            end
  in  streamFold processDef envs defs
  end
```

⟨*more read-eval-print handlers* 330c⟩≡ (330b)

```
| TypeError          msg => continue ("type error: " ^ msg)
| BugInTypeInference msg => continue ("bug in type inference: " ^ msg)
```


BugInType-Inference 294d
elabdef 324d
type env 214
evaldef 327b
mlSyntax 696a
streamFold 650a
type type_env 322a
TypeError 321a
USE 287b
use 222a
type value 287c

7.6.9 Initializing the interpreter

We're now ready to put everything together into a working interpreter.

331a ⟨*initialization* 331a⟩≡ (331c) 331b▷

```
val initialEnvs =
  let fun addPrim ((name, prim, tau), (Gamma, rho)) =
        ( bindtyscheme (name, generalize (tau, freetyvarsGamma Gamma), Gamma)
        , bind (name, PRIMITIVE prim, rho)
        )
      val envs  = foldl addPrim (emptyTypeEnv, emptyEnv) (⟨primops :: 328c⟩ nil)
      val basis = ⟨ML representation of initial basis (automatically generated)⟩
      val defs  = reader mlSyntax noPrompts ("initial basis", streamOfList basis)
      fun errout s = TextIO.output(TextIO.stdErr, s ^ "\n")
  in  readCheckEvalPrint (defs, fn _ => (), errout) envs
  end
```

The function `runInterpreter` takes one argument, which tells it whether to prompt.

331b ⟨*initialization* 331a⟩+≡ (331c) ◁331a

```
fun runInterpreter noisy =
  let fun writeln s = app print [s, "\n"]
      fun errout s = TextIO.output(TextIO.stdErr, s ^ "\n")
      val prompts = if noisy then stdPrompts else noPrompts
      val defs = reader mlSyntax prompts ("standard input", streamOfLines TextIO.stdIn)
  in  ignore (readCheckEvalPrint (defs, writeln, errout) initialEnvs)
  end
```

7.6.10 Putting all the pieces together

We stitch together the parts of the implementation in this order:

331c ⟨*ml.sml* 331c⟩≡

```
⟨environments (generated automatically)⟩
⟨Hindley-Milner types (generated automatically)⟩
⟨lexical analysis (generated automatically)⟩
⟨abstract syntax and values 287a⟩
⟨values (generated automatically)⟩
⟨parsing (generated automatically)⟩
⟨implementation of use (generated automatically)⟩
⟨type inference 322e⟩
⟨checking and evaluation 330a⟩
⟨initialization 331a⟩
⟨command line (generated automatically)⟩
```

7.6.11 The initial basis

The initial basis in μML is almost exactly as in μScheme. The single difference is that in an association list, we have to use a list of pairs, not a list of 2-element lists. This representation makes it possible for keys and values to be of different types.

332a ⟨*additions to the μML initial basis* 332a⟩≡ 332b▷
```
(define list1 (x) (cons x '()))
(define bind (x y alist)
  (if (null? alist)
    (list1 (pair x y))
    (if (= x (fst (car alist)))
      (cons (pair x y) (cdr alist))
      (cons (car alist) (bind x y (cdr alist))))))
```

We need a test to see if a variable is bound; we can't return *nil* for unbound variables, because *nil* is not always of the right type. It must therefore be an error to look up an unbound variable.

332b ⟨*additions to the μML initial basis* 332a⟩+≡ ◁332a
```
(define isbound? (x alist)
  (if (null? alist)
    #f
    (if (= x (fst (car alist)))
      #t
      (isbound? x (cdr alist)))))
(define find (x alist)
  (if (null? alist)
    (error 'not-found)
    (if (= x (fst (car alist)))
      (snd (car alist))
      (find x (cdr alist)))))
```

The rest of the basis is as in μScheme, and it is left for Appendix H.

7.7 Hindley-Milner as it really is

car P 329b
cdr P 329b
cons P 329b
error P 329c
fst P 806c
null? P 329a
pair P 806c
snd P 806c

The Hindley-Milner type system has been used in many languages, but the best known is the one Milner himself worked on: Standard ML. In Standard ML, as in most other languages based on Hindley-Milner, a programmer can mix inferred types with explicit types. For example, in Standard ML one can write explicit type variables after a `val` or `fun` keyword. This syntax is essentially a way of writing `type-lambda`. Standard ML does not have explicit instantiation; instead it has *type constraints*. An explicit instantiation operator says at what type a polymorphic value is instantiated, whereas a type constraint says what is the type of the resulting instance. Here's an example function from Typed μScheme, which has explicit types and `type-lambda`.

332c ⟨*length function for Typed μScheme* 332c⟩≡
```
(val-rec
  (forall ('a) (function ((list 'a)) int))
  length
  (type-lambda ('a)
    (lambda (((list 'a) xs))
      (if ((@ null? 'a) xs) 0
        (+ 1 ((@ length 'a) ((@ cdr 'a) xs)))))))
```

In Standard ML, we write the type parameter after `val` or `fun`, and we use type constraints on lambda-bound and polymorphic variables:

333 ⟨*length function for Standard ML* 333⟩≡

```
val 'a rec length =
  fn (xs : 'a list) =>
    if (null : 'a list -> bool) xs then 0
    else 1 + (length : 'a list -> int) ((tl : 'a list -> 'a list) xs)
```

Different implementations of Standard ML use different notations for the type parameter; Moscow ML puts `'a` before `rec`, but Standard ML of New Jersey puts `'a` after `rec`.

In real languages, the Hindley-Milner type system is just a starting point. The most interesting extensions may be those that are used in the functional language Haskell, which has a clever type system that combines Hindley-Milner type inference and operator overloading. Most implementations of Haskell also support more general polymorphism; for example, the Glasgow Haskell Compiler provides an explicit `forall` syntax that supports lambda-bound variables with polymorphic types.

7.8 Further reading

The original work on the Hindley-Milner type system appears in two papers. Milner (1978) presents the computer-science point of view; Hindley (1969) presents the logician's point of view. Damas and Milner (1982) describe "Algorithm W," the first algorithm used to infer types for ML programs. When I talk about "the method of explicit substitutions," I mean Algorithm W.

Odersky, Sulzmann, and Wehr (1999) present HM(X), a general system for implementing Hindley-Milner type inference with abstract constraints. This system is considerably more ambitious than μML; it allows a very broad class of constraints, and it completely decouples constraint solving from type inference. Pottier and Rémy (2005) use the power of HM(X) to explore a number of extensions to ML. Their tutorial includes code written in the related language OCaml.

Vytiniotis, Peyton Jones, and Schrijvers (2010) argue that as type systems grow more sophisticated, Milner's LET rule makes it harder, not easier to work with the associated constraints. They recommend that by default, the types of LET-bound names should *not* be generalized.

Cardelli (1997) provides a general tutorial on type sytems. Cardelli (1987) has also written a tutorial specifically on type inference; it includes an implementation in Modula-2. The implementation represents type variables using mutable cells and does not use explicit substitutions.

Peyton Jones et al. (2007) show how by adding type annotations, one can implement type inference for types in which a $\forall$ quantifier may appear to the left an arrow, that is, types in which functions may require callers to pass polymorphic arguments. Such types are an example of *higher-rank types*. The authors present very clear synopses of both nondeterministic and deterministic rules for both basic type inference and their extended version. The paper is accompanied by code and will repay careful study.

Material on Haskell can be found at `www.haskell.org`. Jones (1999) presents an implementation, in Haskell, of Haskell's type system.

7.9 Exercises

Learning about the type system

1. *Without using any primitives*, and without using `letrec`, write a function in μML that has type $\forall\alpha, \beta \,.\, \alpha \to \beta$.

2. The type rules of μML permit the following declaration:

   ```
   (val-rec n n)
   ```

   Assuming this declaration were to be permitted, what type scheme would be inferred for `n`?

3. Prove that for any type scheme σ, there is a *most general* instance $\tau <: \sigma$. An instance τ is a most general instance if and only if $\forall\tau' \,.\, \tau' <: \sigma \implies \tau' <: \tau$.

4. Rewrite these type rules to use explicit constraints:

   (a) The rule for LAMBDA:

   $$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e) : \tau_1 \times \cdots \times \tau_n \to \tau} \qquad \text{(LAMBDA)}$$

   (b) The rule for BEGIN:

   $$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n}{\Gamma \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n} \qquad \text{(BEGIN)}$$

5. Prove that in μML, a top-level type environment never contains a free type variable. Your proof should be by induction on the sequence of steps used to create Γ:

   - Prove that an empty type environment contains no free type variables.
   - Using the code in chunk 331a, show that if Γ contains no free type variables, then `addPrim((`x`, `p`, `τ`), (`Γ`, `ρ`))` returns a pair in which the new Γ' also contains no free type variables.
   - Show that if Γ contains no free type variables, and if $\langle \text{VAL}(x, e), \Gamma\rangle \to \Gamma'$, then Γ' contains no free type variables.
   - Show that if Γ contains no free type variables, and if $\langle \text{VAL-REC}(x, e), \Gamma\rangle \to \Gamma'$, then Γ' contains no free type variables.

Learning about constraints

6. Prove that satisfaction is invariant under substitution. That is, prove that if constraint C is satisfied and θ is a substitution, then θC is also satisfied.

7. This exercise asks you to verify the arguments about solving conjunctions that are made in Section 7.5.3.

   (a) Find two constraints C_1 and C_2 and substitutions θ_1 and θ_2 such that
   - C_1 has a free type variable,
   - C_2 has a free type variable,
   - θ_1 solves C_1,
   - θ_2 solves C_2, and
   - $\theta_2 \circ \theta_1$ solves $C_1 \wedge C_2$.

   (b) Find two constraints C_1 and C_2 and substitutions θ_1 and θ_2 such that
   - C_1 has a free type variable,
   - C_2 has a free type variable,
   - θ_1 solves C_1,
   - θ_2 solves C_2, and
   - $\theta_2 \circ \theta_1$ does *not* solve $C_1 \wedge C_2$.

8. This exercise asks you to investigate the relationship between solvability and conjunction.

   (a) Using the proof system in Section 7.5.3, prove that if the constraint $C_1 \wedge C_2$ is solvable, then constraints C_1 and C_2 are also solvable.

   (b) Find a particular pair of constraints C_1 and C_2 such that C_1 is solvable, C_2 is solvable, but $C_1 \wedge C_2$ is *not* solvable. *Prove* that $C_1 \wedge C_2$ is not solvable.

9. Using the definition of what it means for a constraint to be satisfied, prove that the SOLVEDCONJUNCTION rule on page 315 is sound. That is, prove that whenever both premises hold, so does the conclusion. The rule is reproduced here:

$$\frac{\theta_1 C_1 \text{ is satisfied} \qquad \theta_2(\theta_1 C_2) \text{ is satisfied}}{(\theta_2 \circ \theta_1)(C_1 \wedge C_2) \text{ is satisfied}} \qquad \text{(SOLVEDCONJUNCTION)}$$

10. Prove that if τ_2 *mentions* α but is *not equal* to α, then there is no substitution θ such that $\theta\alpha = \theta\tau_2$. (*Hint*: count type constructors.)

11. Using the laws for applying substitutions to constraints and to types, prove that the SOLVECONAPPCONAPP rule on page 316 is sound. That is, prove that a substitution θ solves the constraint

$$\text{CONAPP}(\tau, \langle \tau_1, \ldots, \tau_n \rangle) \doteq \text{CONAPP}(\tau', \langle \tau'_1, \ldots, \tau'_n \rangle)$$

   if and only if it also solves the constraint

$$\tau \doteq \tau' \wedge \tau_1 \doteq \tau'_1 \wedge \cdots \wedge \tau_n \doteq \tau'_n.$$

12. Pages 317–317 discuss the solution of constraint 7.1, which involves synthesizing and solving constraints 7.2 to 7.11. List the numbered constraints *in the order that they would be solved* by a recursive solver.

13. Suppose that you have a function `unify` such that given any two types τ_1 and τ_2, `unify(`τ_1`,` τ_2`)` returns a substitution θ such that $\theta\tau_1 = \theta\tau_2$, or if no such θ exists, raises an exception. Use `unify` to implement a constraint solver.

    Hint: To help convert a constraint-solving problem into a unification problem, try using the SOLVECONAPPCONAPP rule on page 316 *in reverse.*

Learning about the interpreter

14. Using the ideas in Section 7.5.3, implement a function `solve` which takes as argument a constraint of type `con` and returns an idempotent substitution of type `subst`. (If a substitution is created using only the value `idsubst` and the functions `|-->` and `compose`, then it is guaranteed to be idempotent.) The resulting substitution should solve the constraint, obeying the law

    `solves (solve` C`,` C`).`

    If the constraint has no solution, your function should call `unsatisfiableEquality` from chunk 321a, which raises the `TypeError` exception.

15. Implement type inference. That is, complete the definitions of functions `ty` and `literal` so they never raise `LeftAsExercise`. If your code discovers a type error, it should raise the exception `TypeError`.

    The function `literal` should give a suitable type to integer literals, Boolean literals, symbol literals (which have type `sym`), and quoted lists in which all elements have the same type, including the empty list. For example, the value `'(1 2 3)` should have type `int list`. Values created using `CLOSURE` or `PRIMITIVE` cannot possibly appear in a `LITERAL` node, so if your `literal` function sees such a value, it would be justified in raising `BugInTypeInference`.

    You can adapt the `typeof` function from Chapter 6, but you may find it easier to start from scratch. Whatever you do, don't overlook the `typesof` function.

    You will probably find it helpful to refer to the typing rules for μML, which are summarized in Figures 7.2 to 7.5 on pages 339 and 340.

16. Extend μML with primitives `pair`, `fst`, and `snd`. Give the primitives appropriate types. Function `pair` should be used to create pairs of any type, and functions `fst` and `snd` should retrieve the elements of any pair.

17. Improve the error messages issued by the μML interpreter.

    (a) If a type error arises from a function application, show the function and show the arguments. Show what types of arguments the function expects and what the types the arguments actually have.

    (b) If a type error arises from an IF expression, show either that the type of the condition is inconsistent with `bool` or that the two arms do not have consistent types.

(c) Highlight the differences between inconsistent types. For example, if your message says

```
function cons expected int * int list, got int * int
```

this is better than saying "cannot unify `int` and `int list`," but it is not as good as showing *which* argument caused the problem, e.g.,

```
function cons expected int * >>int list<<, got int * >>int<<
```

To complete this part, it would be helpful to define a more sophisticated version of `unsatisfiableEquality`, which returns strings in which types that don't match are highlighted.

To associate a type error with a function application, you will have to look at the APPLY rule on page 306 and check to see whether the constraint in the premise is solvable, and if so, whether the constraint in the conclusion is solvable.

18. Extend μML by adding support for tuple types of arbitrary size. That is, add a new kind of definition (`define-tuple ty f1 f2 ... fn`) such that

- `ty` is a function of `n` arguments that produces an `n`-tuple.
- for `i` from 1 to `n`, `fi` is a function that extracts the `i`th element from an `n`-tuple.

Using `define-tuple`, we could define pairs by writing (`define-tuple pair fst snd`), or triples as follows:

337a ⟨*exercise transcript* 337a⟩≡ 337b▷

```
-> (define-tuple triple one two three)
triple : 'a * 'b * 'c -> 'a * 'b * 'c
one : ('a * 'b * 'c) -> 'a
two : ('a * 'b * 'c) -> 'b
three : ('a * 'b * 'c) -> 'c
```

19. Extend μML by adding support for tuples with generativity. That is, add a new kind of definition (`define-struct ty f1 f2 ... fn`), which is like `define-tuple`, except that the type `ty` is uniquely different from any other type:

337b ⟨*exercise transcript* 337a⟩+≡ ◁337a

```
-> (define-struct key-value key value)
key-value : 'a * 'b -> ('a, 'b) key-value
key : ('a, 'b) key-value -> 'a
value : ('a, 'b) key-value -> 'b
```

Make sure (`fst (key-value 'nr 152)`) does *not* have a type.

20. Extend μML by adding mutation.

    (a) Add new primitives `ref`, `!`, and `:=` with the same meanings as in full ML. You will have to add a case to the definition of `value`, but you should not have to touch environments or the evaluator.

    (b) Write a program in extended μML that subverts the type system, i.e., that causes the evaluator to raise the exception `BugInTypeInference`. For example, try writing a program that applies `+` to a non-integer argument. (Hint: `(ref '())`.)

    (c) Restore type safety by implementing the "value restriction" on polymorphism: a top-level type can be generalized only if it is a syntactic value (variable or λ-abstraction).

21. Change the implementation of type inference so that instead of inferring and checking types in one step, the interpreter takes an untyped term and infers an explicitly typed term in Typed μScheme.

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{Var}\ \frac{\Gamma(x) = \sigma \qquad \tau' <: \sigma}{\Gamma \vdash x : \tau'} \qquad \textsc{If}\ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau}$$

$$\textsc{Apply}\ \frac{\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \qquad \Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \tau}$$

$$\textsc{Lambda}\ \frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e) : \tau_1 \times \cdots \times \tau_n \to \tau}$$

$$\textsc{Let}\ \frac{\begin{array}{c}\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \\ \sigma_i = \texttt{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \le i \le n \\ \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau\end{array}}{\Gamma \vdash \textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau}$$

$$\textsc{Letrec}\ \frac{\begin{array}{c}\Gamma' = \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \\ \Gamma' \vdash e_i : \tau_i, \quad 1 \le i \le n \\ \sigma_i = \texttt{generalize}(\tau_i, \text{ftv}(\Gamma)), \quad 1 \le i \le n \\ \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau\end{array}}{\Gamma \vdash \textsc{letrec}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau}$$

$$\textsc{Letstar}\ \frac{\Gamma \vdash \textsc{let}(\langle x_1, e_1 \rangle, \textsc{letstar}(\langle x_2, e_2, \ldots, x_n, e_n \rangle, e)) : \tau \qquad n > 0}{\Gamma \vdash \textsc{letstar}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau} \qquad \textsc{EmptyLetstar}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textsc{letstar}(\langle\rangle, e) : \tau}$$

$$\textsc{Begin}\ \frac{\Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n}{\Gamma \vdash \textsc{begin}(e_1, \ldots, e_n) : \tau_n} \qquad \textsc{EmptyBegin}\ \frac{}{\Gamma \vdash \textsc{begin}() : \texttt{unit}}$$

Figure 7.2: Nondeterministic typing rules for expressions

$\boxed{\langle d, \Gamma \rangle \to \Gamma'}$

$$\textsc{Val}\ \frac{\Gamma \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \text{ftv}(\Gamma))}{\langle \textsc{val}(x, e), \Gamma \rangle \to \Gamma\{x \mapsto \sigma\}}$$

$$\textsc{ValRec}\ \frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \text{ftv}(\Gamma))}{\langle \textsc{val-rec}(x, e), \Gamma \rangle \to \Gamma\{x \mapsto \sigma\}} \qquad \textsc{Exp}\ \frac{\langle \textsc{val}(\texttt{it}, e), \Gamma \rangle \to \Gamma'}{\langle \textsc{exp}(e), \Gamma \rangle \to \Gamma'}$$

$$\textsc{Define}\ \frac{\langle \textsc{val-rec}(f, \textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e)), \Gamma \rangle \to \Gamma'}{\langle \textsc{define}(f, (\langle x_1, \ldots, x_n \rangle, e)), \Gamma \rangle \to \Gamma'}$$

Figure 7.3: Nondeterministic typing rules for definitions

$\boxed{C, \Gamma \vdash e : \tau}$

$$\textsc{Var}\quad \frac{\Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \qquad \alpha'_1, \ldots, \alpha'_n \text{ are fresh and distinct}}{\mathbf{T}, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \cdots \circ (\alpha_n \mapsto \alpha'_n))\,\tau}$$

$$\textsc{If}\quad \frac{C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3}{C \wedge \tau_1 \doteq \mathtt{bool} \wedge \tau_2 \doteq \tau_3, \Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau_2}$$

$$\textsc{Apply}\quad \frac{C, \Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \doteq \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \alpha}$$

$$\textsc{Let}\quad \frac{\begin{array}{c} C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \doteq \theta\alpha \mid \alpha \in \operatorname{dom}\theta \cap \operatorname{ftv}(\Gamma)\} \\ \sigma_i = \mathtt{generalize}(\theta\tau_i, \operatorname{ftv}(\Gamma) \cup \operatorname{ftv}(C')), \quad 1 \le i \le n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau}$$

$$\textsc{LetRec}\quad \frac{\begin{array}{c} \Gamma' = \Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}, \text{ where all } \alpha_i\text{'s are distinct and fresh} \\ C_r, \Gamma' \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ C = C_r \wedge \tau_1 \doteq \alpha_1 \wedge \cdots \wedge \tau_n \doteq \alpha_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \doteq \theta\alpha \mid \alpha \in \operatorname{dom}\theta \cap \operatorname{ftv}(\Gamma)\} \\ \sigma_i = \mathtt{generalize}(\theta\tau_i, \operatorname{ftv}(\Gamma) \cup \operatorname{ftv}(C')), \quad 1 \le i \le n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \textsc{letrec}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau}$$

Figure 7.4: Constraint-based typing rules for expressions

$\boxed{\langle d, \Gamma\rangle \to \Gamma'}$

$$\textsc{Val}\quad \frac{C, \Gamma \vdash e : \tau \qquad \theta C \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \mathtt{generalize}(\theta\tau, \operatorname{ftv}(\Gamma))}{\langle \textsc{val}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}}$$

$$\textsc{ValRec with constraints}\quad \frac{C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \qquad \alpha \text{ is fresh} \qquad \theta(C \wedge \alpha \doteq \tau) \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \mathtt{generalize}(\theta\alpha, \operatorname{ftv}(\Gamma))}{\langle \textsc{val-rec}(x, e), \Gamma\rangle \to \Gamma\{x \mapsto \sigma\}}$$

Figure 7.5: Constraint-based typing rules for definitions