Chapter 14

Type Reconstruction

The faculty of deduction is certainly contagious ...

— *Sherlock Holmes in* The Problem of Thor Bridge
by Sir Arthur Conan Doyle

14.1 Introduction

In the variants of FL/X that we've studied so far, it is necessary to specify explicit type information in certain situations. All variables introduced by a `lambda` must be explicitly typed, for instance. Not all type information needs to be explicitly declared, however. For example, the return type of a procedure need not be explicitly declared.

What determines the placement of explicit type information in a language? That is, why does some type information have to be provided while other type information can be elided? The answer to this question lies in the structure of the type checker. As noted earlier, a simple type checker has the structure of an evaluator. Consider the type checking of a `lambda` expression. When entering a `lambda` expression, the type checker has no information about the types of the formal parameters; these must be provided explicitly. However, once the types of the formals are known, it is easy for the type checker to determine the type of the body, so this information need not be declared.

Could a more sophisticated type checker do its job with even less explicit information? Certainly, programmers can reason proficiently about type information in many programs where there are no explicit types at all. Such reasoning is important because understanding the type of an expression, especially one

that denotes a procedure, is often a major step in figuring out what purpose the expression serves in the program. As an example of this kind of type reasoning, consider the following expression:

```
(lambda (f x y)
  (if (f x y) (f 3 y) (f x "twenty-three"))))
```

By studying the various ways in which `f`, `x`, and `y` are used in the body of the above `lambda` expression, we can piece together a lot of information about the types of these variables. The application `(f x y)`, for example, returns a boolean because it is used as the predicate in an `if` expression. Thus, `f` is a procedure of two arguments that returns a boolean. Since both branches of the `if` expression are calls to `f`, we know that the procedure created by the `lambda` expression must also return a boolean. In fact, looking at the consequent and alternative of the `if`, we can even say more about `f`: its first argument is a number and its second argument is a string. Thus, `x` must be a number and `y` must be a string.

There is no reason that a program cannot carry out the same kinds of reasoning exhibited above. Automatically computing the type of an expression that does not contain type information is known as **type reconstruction** or **type inference**. Type reconstruction is more complicated than type checking because type reconstruction must operate properly without programmer supplied type assertions.

Type reconstruction is the formalization of the kind of reasoning seen in the example above. A type reconstruction algorithm is an automatic way of determining the types of an expression (and all the subexpressions along the way). We can think of the different subexpressions in the above example as specifying constraints on the types of the expressions. It is possible to view these constraints as a set of simultaneous type equations that restrict the type of an expression. If these equations cannot be solved, then the expression is not well-typed. If these equations can be solved, then the **most general** typing for the expression results. In the event that several types may be assigned to an expression, then the most general type is the type, T, such that all other possible types are substitution instances of T. I.e., for any type, S, the expression may have, there is a substitution that can be applied to the most general type to get S. The type system of a type reconstructed language is usually designed so that there is a unique most general type for every typable expression. Most general types are often called **principal types**.

Consider the `lambda` expression studied above. Suppose that[1]

[1]Here we denote unknowns in the equations by names prefixed with a question mark, to

- `?l` is the type of the result of evaluating the `lambda` expression.
- `?i` is the type of the result of evaluating the `if` expression.
- `?f` is the type of `f`.
- `?x` is the type of `x`.
- `?y` is the type of `y`

Then the equations that are implied by the expression are

```
?l  =  (-> (?f ?x ?y) ?i)
?f  =  (-> (?x ?y) ?a)
?a  =  bool
?f  =  (-> (int ?y) ?b)
?f  =  (-> (?x string) ?c)
?b  =  ?c
?i  =  ?b
```

A solution to the above equations yields the following variable bindings:

```
?a  =  ?b = ?c = ?i = bool
?x  =  int
?y  =  string
?f  =  (-> (int string) bool)
?l  =  (-> ((-> (int string) bool) int string) bool)
```

Note that a system of type equations need not always have the neat form of solution indicated by the example. For example, the system associated with

```
(lambda (f x y)
  (if (f x y) (f 3 y) (f y "seven")))
```

has no solution since it is overconstrained: the `int` and `string` types are disjoint. On the other hand, the system may be underconstrained, as in the following perturbation of the example:

distinguish them from variables in the language.

```
(lambda (h x y)
  (if (h x y) (h x y) (h x "seven")))
```

In this case, the type of `x` is unknown, and the type deduced for the expression is

```
(-> ((-> (?x string) bool) ?x string) bool)
```

The appearance of an unknown type variable in this type is the way that this particular type language indicates the potential for polymorphism. We will see below under what conditions such a type is viewed as polymorphic. In other notations we have seen, a polymorphic type would be expressed as:

```
(forall (t) (-> ((-> (t string) bool) t string) bool))
```

or

$$\forall t\,.\ \texttt{((string t)} \rightarrow \texttt{bool)} \times \texttt{t} \times \texttt{string} \rightarrow \texttt{bool}$$

14.2 A Language with Type Reconstruction: FL/R

In this section, we'll consider issues in type reconstruction for a variant of FL called FL/R. The grammar for FL/R is given in Figure 14.1. Note that there are no explicit type declarations in the expressions of the language.

Figure 14.2 contains the typing rules for FL/R. The rules for literals, conditionals, abstractions, and applications are similar to the ones for FL/X. The only difference is in the `lambda` rule. Whereas the FL/X typing rule for `lambda` uses the explicit type declarations for each of the variables, the `lambda` rule in FL/R "guesses" the types of the variables, and then checks to see that its guess is correct. The typing rules do not explain how these guesses are made. The details of guessing will be specified by the reconstruction algorithm presented in Section 14.4 below.

The typing rules for `let`, `letrec` and variables contain some new ideas. The motivation for these concepts is that we'd like type reconstruction to be able to reconstruct polymorphic types, at least in some simple cases. As an example of where we'd like to infer a polymorphic type, consider:

```
(let ((f (lambda (x) x)))
  (if (f #t) (f 1) (f 2)))
```

Here, we would like `f` to have the type `(-> (bool) bool)` when applied to `#t`, and the type `(-> (int) int)` when applied to `1` and `2`. If we required each variable to have only one type associated with it, this kind of polymorphic behavior would not be allowed in the language.

$P \in \text{Program}_{FL/R}$ $I \in \text{Identifier}_{FL/R}$ = usual identifiers
$E \in \text{Exp}_{FL/R}$ $B \in \text{Boollit}_{FL/R} = \{\texttt{\#t}, \texttt{\#f}\}$
$AB \in \text{Abstraction}_{FL/R}$ $N \in \text{Intlit}_{FL/R} = \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$L \in \text{Lit}_{FL/R}$ $O \in \text{Primop}_{FL/R}$
$T \in$ Type
$TS \in$ Type-schema

P ::= `(flr (`I_{fml}`*) `E_{body} `(define `I_{name} E_{defn}`)*)`

E ::= L | I | `(error `I`)`
| `(if `E_{test} E_{then} E_{else}`)` | `(begin `E^+`)`
| `(lambda (`I_{fml}`*) `E_{body}`)` | `(`E_{rator} E_{rand}`*)` | `(primop `O_{op} E_{arg}`*)`
| `(let ((`I_{name} E_{defn}`)*) `E_{body}`)` | `(letrec ((`I_{name} E_{defn}`)*) `E_{body}`)`

L ::= `#u` | B | N | `(symlit `I`)`

$O_{FL/R}$ ::= `+` | `-` | `*` | `/` | `%` [Arithmetic ops]
| `<=` | `<` | `=` | `!=` | `>` | `>=` [Relational ops]
| `not` | `band` | `bor` [Logical ops]
| `pair` | `fst` | `snd` [Pair ops]
| `cons` | `car` | `cdr` | `null` | `null?` [List ops]
| `cell` | `^` | `:=` [Mutable cell ops]

T ::= `unit` | `int` | `bool` | `sym` [Base Types]
| I [Type Variable]
| `(->` (`$T$`*) T_{body}`)` [Arrow Type]
| `(pairof` T_1 T_2`)` [Pair Type]
| `(listof` T`)` [List Type]
| `(cellof` T`)` [Cell Type]

TS ::= `(generic (`I`*) `T`)` [Type Schema]

Figure 14.1: Grammar for FL/R

$$\vdash \texttt{\#u} : \texttt{unit} \qquad [unit]$$

$$\vdash B : \texttt{bool} \qquad [bool]$$

$$\vdash N : \texttt{int} \qquad [int]$$

$$\vdash \texttt{(symbol } I\texttt{)} : \texttt{sym} \qquad [symbol]$$

$$[\ldots, I\texttt{:}T, \ldots] \vdash I : T \qquad [var]$$

$$[\ldots, I\texttt{:(generic (}I_1 \ \ldots \ I_n\texttt{) } T_{body}\texttt{)}, \ldots] \vdash I : ([T_i/I_i]_{i=1}^{n})T_{body} \qquad [genvar]$$

$$A \vdash \texttt{(error } I\texttt{)} : T \qquad [error]$$

$$\frac{A \vdash E_{test} : \texttt{bool} \quad ; \quad A \vdash E_{con} : T \quad ; \quad A \vdash E_{alt} : T}{A \vdash \texttt{(if } E_{test} \ E_{con} \ E_{alt}\texttt{)} : T} \qquad [if]$$

$$\frac{\forall_{i=1}^{n} \ . \ A \vdash E_i : T_i}{A \vdash \texttt{(begin } E_1 \ \ldots \ E_n\texttt{)} : T_n} \qquad [begin]$$

$$\frac{A[I_1 : T_1\texttt{,} \ \ldots\texttt{,} \ I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{) } E_{body}\texttt{)} : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T_{body}\texttt{)}} \qquad [\lambda]$$

$$\frac{\begin{array}{c} A \vdash E_{rator} : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T_{body}\texttt{)} \\ \forall_{i=1}^{n} \ . \ A \vdash E_i : T_i \end{array}}{A \vdash \texttt{(}E_{rator} \ E_1 \ \ldots \ E_n\texttt{)} : T_{body}} \qquad [apply]$$

$$\frac{\begin{array}{c} A_{standard} \vdash O : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T\texttt{)} \\ \forall_{i=1}^{n} \ . \ A \vdash E_i : T_i \end{array}}{A \vdash \texttt{(primop } O \ E_1 \ \ldots \ E_n\texttt{)} : T} \qquad [primop]$$

$$\frac{\begin{array}{c} \forall_{i=1}^{n} \ . \ A \vdash E_i : T_i \\ A[I_1\texttt{:}\mathtt{GenPure}\texttt{(}E_1\texttt{,} \ T_1\texttt{,} \ A\texttt{)},\ldots,I_n\texttt{:}\mathtt{GenPure}\texttt{(}E_n\texttt{,} \ T_n\texttt{,} \ A\texttt{)}] \vdash E_{body} : T_{body} \end{array}}{A \vdash \texttt{(let ((}I_1 \ E_1\texttt{) } \ldots \ \texttt{(}I_n \ E_n\texttt{))) } E_{body}\texttt{)} : T_{body}} \qquad [let]$$

$$\frac{\begin{array}{c} \forall_{i=1}^{n} \ . \ A[I_1 : T_1\texttt{,} \ \ldots\texttt{,} \ I_n : T_n] \vdash E_i : T_i \\ A[I_1\texttt{:}\mathtt{GenPure}\texttt{(}E_1\texttt{,} \ T_1\texttt{,} \ A\texttt{)},\ldots,I_n\texttt{:}\mathtt{GenPure}\texttt{(}E_n\texttt{,} \ T_n\texttt{,} \ A\texttt{)}] \vdash E_{body} : T_{body} \end{array}}{A \vdash \texttt{(letrec ((}I_1 \ E_1\texttt{) } \ldots \ \texttt{(}I_n \ E_n\texttt{))) } E_{body}\texttt{)} : T_{body}} \qquad [letrec]$$

$$\frac{\begin{array}{c} A_{standard}[I_1 : T_1\texttt{,} \ \ldots\texttt{,} \ I_n : T_n] \vdash \\ \texttt{(letrec ((}I_{d_1} \ E_1\texttt{) } \ldots \ \texttt{(}I_{d_k} \ E_k\texttt{)) } E_{body}\texttt{)} : T \end{array}}{\begin{array}{c} \vdash_{prog} \texttt{(flr (}I_1 \ \ldots \ I_n\texttt{) } E_{body} \ \texttt{(define } I_{d_1} \ E_1\texttt{) } \ldots \ \texttt{(define } I_{d_k} \ E_k\texttt{))} \\ : \texttt{(->} \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T\texttt{)} \end{array}} \qquad [program]$$

$$Gen(T, A) = \texttt{(generic (}I_1 \ \ldots \ I_n\texttt{) } T\texttt{)}, \text{where } \{I_i\} = FTV(T) - FTE(A)$$

$$GenPure(E, T, A) = \begin{array}{l} Gen(T, A) \text{ if } E \text{ is pure} \\ T \text{ otherwise} \end{array}$$

Figure 14.2: Typing rules for FL/R.

In order to handle this simple polymorphism, we introduce the notion of a **type schema** (*TS* in Figure 14.1). A type schema is a pattern for a type expression that is abstracted over variables; the schema can be instantiated by binding the variables to particular types. For example, a type schema for the identity procedure is

```
(generic (t) (-> (t) t))
```

Type schemas for various list operations are shown below:

```
cons:   (generic (t) (-> (t (list-of t)) (list-of t)))
car:    (generic (t) (-> ((list-of t)) t))
cdr:    (generic (t) (-> ((list-of t)) (list-of t)))
null?   (generic (t) (-> ((list-of t)) bool))
null    (generic (t) (-> () (list-of t)))
```

Unlike the `forall` type of FL/X, a type schema cannot appear as a subexpression of a type expression. That is, according to the grammar for type schemas, `generic` can only appear once, at the outermost level. A type schema thus represents types that are universally quantified over a set of variables.

The reason for this restriction on type schemas is that type reconstruction is greatly complicated in the more general case. In some cases, it is unknown whether it can even be accomplished; other situations involving general polymorphic types have been proven undecidable. Reconstruction using type schemas, though, is decidable. Type reconstruction based on type schemas is usually called **Hindley-Milner** type reconstruction, after its inventors.

FL/R's use of type schemas means that certain meaningful FL programs are not well-typed in FL/R, even if they could be assigned types in FL/X. Consider the following program:

```
(lambda (f)
  (if (f #t) (f 1) (f 2)))
```

Intuitively, the type for this procedure is something like

```
(-> ((generic (t) (-> (t) t))) int)
```

But this is not a legal type or type schema. The closest legal type schema we could make would be:

```
(generic (t) (-> ((-> (t) t)) int))
```

Although this bears some similarity to the desired type, it is not correct. For example, instantiating the schema with `t` bound to `int` would lead us to believe that the above procedure could take the integer successor procedure as its argument. But this yields a type error, since we shouldn't be able to apply the successor procedure to `#t`.

Type schemas are introduced into the language via `let` and `letrec`. The idea is that any new type variables introduced by the bindings (free type variables that do not appear already in the type environment, A) can be viewed as generic. For example,

```
(let ((f (lambda (x) x)))
  (if (f true) (f 3) (f 4)))
```

will now type check, because the type checker will guess that `(lambda (x) x)` has the type `(-> (t13) t13)` where `t13` is a newly minted type variable. Since this type variable is not constrained by information imported into the expression `(lambda (x) x)`, it can be generalized, and thus `f` can have a different type each time it is used (via the [*genvar*]). How these names are guessed will become clearer when we present a type reconstruction algorithm below.

One way to view this type of polymorphism is to imagine that the right hand sides of the `let` or `letrec` bindings are substituted for the identifiers to which they are bound in the body. This would allow the expression to have different types for any new type variables at each use. However, this transformation is only legitimate if the expressions are referentially transparent as discussed in Section 8.2.5.

The [*let*] and [*letrec*] rules restrict polymorphic values to be pure expressions, just as we restricted the body of a `plambda` to be pure, and for the same reason. We will see a more general way to introduce effect restrictions in Chapter 16, but for now, we will insist that E is pure only if it is a syntactic value, i.e., E is a literal, variable reference (note that FL/R does not have mutable variables), a `lambda` expression, or an `if`/`let`/`letrec` all of whose components are syntactic values. In other words, applications or compound expressions (except `lambda`) is not a syntactic value.

14.3 Unification

In order to solve type equations, we will use the **unification** method due to Robinson. Unification takes two types and attempts to find a substitution for special unification variables in the two types (here prefixed with `?`) such that the two expressions are equal. A substitution is a structure that represents constraints between unification variables. It is like a type environment in that it contains bindings of variables (in this case, unification variables, not expression variables) to types. These types may contain other unification variables. A substitution can be applied to a type or expression; this returns a new type or expression in which each unification variable has been replaced by the element to which it is bound in the substitution. A substitution S is said to be **more**

general than a substitution S' (written $S > S'$) if there exists an S'' such that $S' = (S'' \circ S)$.

More formally, a unification algorithm U unifying types T_1 and T_2 with respect to a substitution S (written $U(T_1, T_2, S)$) produces the most general substitution $S' < S$ such that $(S'\ T_1) = (S'\ T_2)$, where the notation $(S\ T)$ designates the result of applying substitution S to the type T. Unification can of course fail. We will represent the result of a failing unification by the token `fail`.

Here are some examples of unification. (Assume that S_0 is the empty substitution, i.e., one not specifying constraints on any variables.)

$$U(\texttt{(?x ?y)}, \texttt{(int ?x)}, S_0) = \{\texttt{?x = int, ?y = int}\}$$

$$U(\texttt{(-> (int) bool)}, \texttt{(-> (bool) ?x)}, S_0) = \texttt{fail}$$

$$U(\texttt{(-> (?x) (-> (?x) ?y))}, \texttt{(-> (int) ?z)}, S_0)$$
$$= \{\texttt{?x = int, ?z = (-> (int) ?y)}\}$$

Note in the examples how the same variable can be used in the two expressions being unified to express a constraint between them. For example, in $U(\texttt{(?x ?y)}, \texttt{(int ?x)}, S_0)$ the variable `?x` is used to say "the first element of the first pair must be the same as the second element of the second pair." This idea is also very important in logic programming, and, in fact, unification lies at the heart of both logic programming and type reconstruction.

14.4 A Type Reconstruction Algorithm

We now present a type reconstruction algorithm similar to one developed by Milner. This algorithm is the basis of type reconstruction in FX, ML, and HASKELL.

This is used (via input) both by handouts 41 and 44 We shall use the following notation to describe the steps of the algorithm:

$$(R[\![E]\!]\ A\ S) = \langle T, S' \rangle$$

The way to read this notation is "Reconstructing the type of expression E in type environment A with respect to substitution S yields the type T and the new substitution S'." Reconstruction may not always succeed; if it is not possible to perform type reconstruction, then

$$(R[\![E]\!]\ A\ S) = \texttt{fail}$$

The algorithm is defined such that the following relationship is satisfied:

$$(\textit{subst-in-type-env}\ S'\ A) \vdash E : (S'\ T)$$

where $(S'\ T)$ means the result of applying the substitution S' to the type T, and *subst-in-type-env* takes a substitution and a type environment, and returns a new type environment in which the substitution has been applied to all the types bound in the environment.

The type of an FL/R expression E can be found by the function *reconstruct*,

$$\mathit{reconstruct}(E) = (\textbf{let}\ \langle T, S\rangle\ \textbf{be}\ \ (R[\![E]\!]\ \ A_0\ \ S_0)\ \textbf{in}\ (S\ \ T))$$

where A_0 is the standard type environment (presumably containing the types for all names in the standard value environment) and S_0 is the empty substitution. Recall that the metalanguage notation

$$\textbf{let}\ \langle T, S\rangle\ \textbf{be}\ \ E_{val}\ \textbf{in}\ E_{body}$$

is a destructuring form of **let** . We assume for simplicity of presentation that **let** propagates failure as well. That is, if the result of evaluating E_{val} is `fail`, then **let** returns `fail` immediately, without evaluating E_{body}.

Figures 14.3 and 14.4 present the algorithm. The handling of literals, conditionals, abstractions, and applications are fairly straightforward. The handling of `let` and `letrec` is complicated by the desire to handle polymorphism. The functions *RgenPure* and *Rgen* are like the *GenPure* and *Gen* functions encountered before, except that they take a substitution as an additional argument. This substitution is applied to both the type and the type environment:

$$\mathit{RgenPure}(E,\ T,\ A,\ S) = \begin{cases} T & \text{if } E \text{ is not pure} \\ \mathit{Rgen}(T, A, S) & \text{otherwise} \end{cases}$$

$\mathit{Rgen}(T,\ A,\ S)$
$\quad = \mathit{Gen}((S\ \ T),$ `(subst-in-type-env` $S\ \ A$`)`$)$
$\quad =$ `(generic (`$J_1\ \ldots\ J_n$`)` $(S\ \ T)),$
$\qquad$ where $\{J_i\} = \mathit{FTV}((S\ \ T)) - \mathit{FTE}((\mathit{subst\text{-}in\text{-}type\text{-}env}\ S\ A))$

Here, *FTV* gives the free type variables of a type expression (i.e., those type variables that are not bound by `generic`), and *FTE* gives the free type variables of a type environment (i.e., all type variables that appear free in some type bound in the environment). A type variable J is a name prefixed with a `?`.

14.5 Discussion

Milner proved two theorems about his type reconstruction algorithm:

1. The semantic soundness theorem states that if an expression is well-typed (by his definition of well-typed, which is expressed as a set of reconstruction rules), then the expression cannot encounter a dynamic type error.

$(R[\![\texttt{\#u}]\!]\ A\ S) = \langle \texttt{unit}, S\rangle$
$(R[\![B]\!]\ A\ S) = \langle \texttt{bool}, S\rangle$
$(R[\![N]\!]\ A\ S) = \langle \texttt{int}, S\rangle$
$(R[\![S]\!]\ A\ S) = \langle \texttt{string}, S\rangle$
$(R[\![\texttt{(symbol } I\texttt{)}]\!]\ A\ S) = \langle \texttt{sym}, S\rangle$
$(R[\![I]\!]\ A[\ldots, I\texttt{:}T, \ldots]\ S) = \langle T, S\rangle$
$(R[\![I]\!]\ A[\ldots, I\texttt{:(generic (}I_1\ \ldots\ I_n\texttt{) }T\texttt{)}, \ldots]\ S) = \langle([\texttt{?v}_i/I_i]_{i=1}^{n})T, S\rangle$,
where $\texttt{?v}_i$ are fresh
$(R[\![I]\!]\ A\ S) = \texttt{fail}$,
$\texttt{where } I \texttt{ unbound in } A$
$(R[\![\texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)}]\!]\ A\ S) =$
let $\langle T_{test}, S_{test}\rangle$ **be** $(R[\![E_{test}]\!]\ A\ S)$ **in**
let $S_{test}{}'$ **be** $U(T_{test}, \texttt{bool}, S_{test})$ **in**
let $\langle T_{con}, S_{con}\rangle$ **be** $(R[\![E_{con}]\!]\ A\ S_{test}{}')$ **in**
let $\langle T_{alt}, S_{alt}\rangle$ **be** $(R[\![E_{alt}]\!]\ A\ S_{con})$ **in**
let $S_{alt}{}'$ **be** $U(T_{con}, T_{alt}, S_{alt})$ **in**
$\langle T_{alt}, S_{alt}{}'\rangle$
$(R[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) }E_{body}\texttt{)}]\!]\ A\ S) =$
let $\langle T_{body}, S_{body}\rangle$ **be** $(R[\![E_{body}]\!]\ A[I_1\texttt{:?v}_1\ \ldots\ I_n\texttt{:?v}_n]\ S)$ **in**
$\langle\texttt{(->(?v}_1\ \ldots\ \texttt{?v}_n\texttt{) }T_{body}\texttt{)}, S_{body}\rangle$,
where $\texttt{?v}_i$ are fresh
$(R[\![\texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)}]\!]\ A\ S) =$
let $\langle T_{rator}, S_{rator}\rangle$ **be** $(R[\![E_{rator}]\!]\ A\ S)$ **in**
let $\langle T_1, S_1\rangle$ **be** $(R[\![E_1]\!]\ A\ S_{rator})$ **in**
$\vdots$
let $\langle T_n, S_n\rangle$ **be** $(R[\![E_n]\!]\ A\ S_{n-1})$ **in**
let S_{final} **be** $U(T_{rator}, \texttt{(->(}T_1\ \ldots\ T_n\texttt{) ?v)}, S_n)$ **in**
$\langle\texttt{?v}, S_{final}\rangle$,
where ?v is fresh.

Figure 14.3: Type reconstruction algorithm for FL/R, Part I.

$$
\begin{array}{l}
(R[\![\texttt{(let ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{)) }E_{body}\texttt{)}]\!]\ A\ S) = \\
\quad \textbf{let}\ \langle T_1, S_1\rangle\ \textbf{be}\ (R[\![E_1]\!]\ A\ S)\ \textbf{in} \\
\qquad \vdots \\
\quad \textbf{let}\ \langle T_n, S_n\rangle\ \textbf{be}\ (R[\![E_n]\!]\ A\ S_{n-1})\ \textbf{in} \\
\qquad R(E_{body}, \\
\qquad\quad A[I_1 : \texttt{RgenPure}(E_1,\ T_1,\ A,\ S_n),\ \ldots,\ I_n : \texttt{RgenPure}(E_n,\ T_n,\ A,\ S_n)], \\
\qquad\quad S_n) \\
(R[\![\texttt{(letrec ((}I_1\ E_1\texttt{) ... (}I_n\ E_n\texttt{)) }E_{body}\texttt{)}]\!]\ A\ S) = \\
\quad \textbf{let}\ A_1\ \textbf{be}\ A\texttt{[}I_1\texttt{:?v}_1\texttt{, ..., }I_n\texttt{:?v}_n\texttt{]}\ \textbf{in} \\
\quad\ \textbf{let}\ \langle T_1, S_1\rangle\ \textbf{be}\ (R[\![E_1]\!]\ A_1\ S)\ \textbf{in} \\
\qquad \vdots \\
\quad\ \textbf{let}\ \langle T_n, S_n\rangle\ \textbf{be}\ (R[\![E_n]\!]\ A_1\ S_{n-1})\ \textbf{in} \\
\quad\ \textbf{let}\ S_B\ \textbf{be}\ U(\texttt{(?v}_1\ \ldots\ \texttt{?v}_n\texttt{)}, \texttt{(}T_1\ \ldots\ T_n\texttt{)}, S_n)\ \textbf{in} \\
\qquad R(E_{body}, \\
\qquad\quad A[I_1 : \texttt{RgenPure}(E_1,\ T_1,\ A,\ S_B),\ \ldots,\ I_n : \texttt{RgenPure}(E_n,\ T_n,\ A,\ S_B)], \\
\qquad\quad S_B) \\
\text{where } \texttt{?v}_i \text{ are fresh}
\end{array}
$$

Figure 14.4: Type reconstruction algorithm for FL/R, Part II.

More generally, he showed that, if with respect to type environment A, an expression E has static type T (i.e., $A \vdash E : T$), then the denotation of E with respect to an environment e that respects A has type T (i.e., $(\mathcal{E}[\![E]\!]\ e) : T$). An environment e respects type environment A if for all variables `x` in A, $(e$ `x`$)$ has type $(A$ `x`$)$.

2. The syntactic soundness theorem states that if the type reconstruction algorithm R above discovers a type for an expression E, then the type it discovers is a provable type of E, and thus E is well-typed.

Of course there are limitations to type reconstruction. For example, in Milner's type system the following expressions are not well-typed:

```
(lambda (x) (x x))                ; Self-application

(lambda (f) (cons (f 1) (f #t))) ; First-class polymorphic values
```

The second restriction appears to be more severe than the first. In fact, there is presently no way of giving an independent characterization of the expressions that are not well-typed in Milner's system.

It is important to note that the kinds of types we can infer are closely related to the kinds of inference rules that we are using. We are often willing to reduce

the power of our type inference system (in terms of the range of types that can be inferred) for an increased simplicity in the type inference rules. Simpler rules are easier to prove sound; they generally imply an easier-to-implement type-inference algorithm as well.

For example, Milner's type-inference algorithm does not handle subtyping in any way. As exhibited by the above example, we assume that the types of procedure arguments in different calls within the same type environment must be exactly the same; we will not search for some "least upper bound" of the two in some type lattice. Similarly, we assume that both branches of an `if` expression have exactly the same type; no subtypting is allowed here either. This means that the type equation solver need only deal with strict equality constraints. The standard unification algorithm is very good at solving such equality constraints. But the minute subtyping is added to type inference, inequality constraints are introduced and the standard unification algorithm doesn't work any more. This doesn't necessarily mean that inference with subtyping is impossible; it's just a lot more complex.

A key advantage of Milner's approach to type inference is that it is decidable. If we try to make a type inference system more powerful by including features like first-class polymorphism, type inference may become undecidable. Here's a table of what is currently known about the decidability of various type inference schemes:

Type of Inference	First-class Polymorphism?	User Declarations	Decidability
Hindley-Milner	no	optional	decidable
Full 2nd-order λ-calculus	yes	none	undecidable
Full 2nd-order λ-calculus	yes	optional	undecidable
Full 2nd-order λ-calculus	yes	non-ML types declared	decidable
Full 2nd-order λ-calculus	yes	required	decidable

▷ **Exercise 14.1** After Alyssa P. Hacker finished her semantics for `producer` and `consumer` from Exercise 9.5, she realized that she also needed to specify typing rules for the new language constructs. Alyssa started by adding the new `producer-of` type to describe producer values:

T ::= ... | (`producer-of` T_{yield} T_{return})

A producer of type (`producer-of` T_{yield} T_{return}) yields values of type T_{yield}; if no more values are to be yielded, it returns a value of type T_{return}.

a. What is the type of the identifier `sum` defined in the following example?

```
(define sum
  (lambda (prod-fn)
    (let ((ans (cell 0)))
      (if (consume (prod-fn #u) n
            (cell-set! ans (+ n (cell-ref ans))))
          (cell-ref ans)
          -1))))
```

b. What are the typing rules for the `producer` and `consumer` constructs?

c. What are the type reconstruction algorithm clauses for `producer` and `consumer`?

◁

▷ **Exercise 14.2** Sam Antix realizes that FL/R supports only homogeneous compound datatypes. He decides to extend FL/R with heterogeneous values called **tuples**. An n-tuple is a value that contains n component values, all of which may have different types. Sam extends the syntax of FL/R as follows:

E ::= ... | `(tuple` E_1 ... E_n`)` | `(tuple-ref` E_{tuple} N_{index} N_{size}`)`

Here is an informal description of Sam's new expressions:

- `(tuple` E_1 ... E_n`)` packages up the values E_1 ... E_n into an n-tuple. Unlike lists and vectors, tuples are heterogeneous data structures: the values of the E_i expressions can all be of different types.
- `(tuple-ref` E_{tuple} N_{index} N_{size}`)` evaluates E_{tuple}, which should be an N_{size}-tuple t, and returns the N_{index}-th component of t. For example:

  ```
  (tuple-ref (tuple 17 #t (symbol captain) "abstraction") 2 4)
  ```

  yields the second element, `#t`, from a 4-tuple. In Sam's syntax, note that the index and size *must* be integer literals — they are *not* general expressions to be evaluated.

a. Extend the FL/R type grammar to handle tuples.

b. Give the typing rules for `tuple` and `tuple-ref`.

c. Specify the type reconstruction algorithm clauses for `tuple` and `tuple-ref`.

d. Louis Reasoner thinks that the form of `tuple-ref` is unwieldy. "I don't see why N_{size} is at all necessary," he complains. "Why don't you make `tuple-ref` have the following form instead?"

`(tuple-ref` E_{tuple} N_{index}`)`

Briefly explain why Louis' suggestion would complicate type reconstruction for FL/R.

e. After Sam successfully explains to Louis his rationale for N_{size}, Louis has another suggestion: "I don't see why the form of N_{index} has to be so restricted. Why not change the form of `tuple-ref` to be the following?"

$$\texttt{(tuple-ref } E_{tuple} \ E_{index} \ N_{size}\texttt{)},$$

where the component index is the computed value of E_{index} rather than a literal integer value.

Why is this a bad idea?

◁

▷ **Exercise 14.3** Ben Bitdiddle enhanced FL/R with several parallel programming constructs. His most important extension is a new construct called `go` that executes multiple expressions in parallel. Ben extended the FL/R grammar as follows:

E ::= ... | `(go (`I_1 ... I_n`)` E_1 ... E_m`)` | `(talk!` I E`)` | `(listen` I`)`

Here is the informal semantics of the newly added constructs: `go` terminates when all of E_1 ... E_m terminate; it returns the value of E_1. `go` includes the ability to use communication variables I_1 ... I_n in a parallel computation. A communication variable can be assigned a value by `talk!`. An expression in `go` can wait for a communication variable to be given a value with `listen`. `listen` returns the value of the variable once it is set with `talk!`. For a program to be well-typed, all E_1 ... E_m in `go` must be well-typed.

Ben extended the type grammar of FL/R as follows:

T ::= ... | `(commof` T`)`

Communication variables will have the unique type `(commof` T`)` where T is the type of value they hold. This will ensure that only communication variables can be used with `talk!` and `listen`, and that communication variables can not be used in any other expression. Ben has already written the typing rules for `talk!` and `listen`:

$$\frac{A \vdash E : \texttt{T} \qquad A \vdash I : \texttt{(commof T)}}{A \vdash \texttt{(talk!} \ I \ E\texttt{)} : \texttt{unit}} \quad [\textit{talk!}]$$

$$\frac{A \vdash I : \texttt{(commof T)}}{A \vdash \texttt{(listen} \ I\texttt{)} : \texttt{T}} \quad [\textit{listen}]$$

a. Give the typing rule for `go`.

b. Give the FL/R reconstruction algorithm clauses for `talk!`, `listen`, and `go`.

◁

Reading

Type reconstruction in programming languages is due to Milner [Mil78], who reinvented work previously done in logic by Curry and by Hindley. Examples of programming languages with type reconstruction are ML [MTH90, MT91], Haskell [HJW+92], and FX [GJSO92].