Polymorphic Type Inference

Michael I. Schwartzbach
`http://www.daimi.au.dk/~mis`

March 1995

Preface

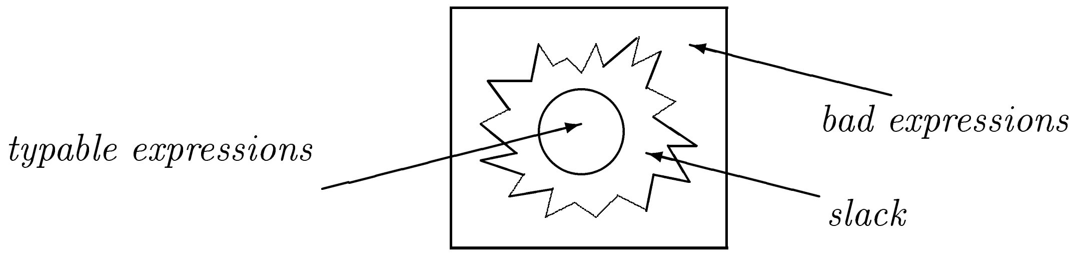
In this lecture we will present a tiny functional language and gradually enrich its type system. We shall cover the basic Curry-Hindley system and Wand's constraint-based algorithm for monomorphic type inference; briefly observe the Curry-Howard isomorphism and notice that logical formalism may serve as the inspiration for new type rules; present the polymorphic Milner system and the Damas-Milner algorithm for polymorphic type inference; see the Milner-Mycroft system for polymorphic recursion; and sketch the development of higher type systems. We will touch upon the relationship between types and logic and show how rules from logic may give inspiration for new type rules. En route we shall encounter the curious discovery that two algorithmic problems for type systems, which have been implemented in popular programming languages, have turned out to be respectively complete for exponential time and undecidable.

1 Type Checking and Type Inference

Two main reasons for introducing types into a programming language are *safety* and *readability*.

Let us first consider the safety aspect. Not all expressions in an untyped language are thought to be sensible. For example, computing the quotient of a text and a function or the square root of a banana is plain nonsense. Let us agree upon the existence of a subset of *bad expressions*, the run-time behaviors of which are undesirable or undefined. Any non-trivial choice of badness leads to an uncomputable subset. Thus, the compiler cannot distinguish the good from the bad.

To ensure safety of programs we introduce a type system and reluctantly sacrifice some good expressions. This happens because we require our type system to ensure that only good expressions are typable. However, since type checking must be decidable, it is clear that some good expressions are not typable. The following diagram illustrates this situation.



The *slack* in a type system is the set of good expressions that it unfairly rejects. The desire to minimize this slack is a driving force in the development of type systems, which leads towards ever more complex type rules. This trend is curbed by the fact that the type discipline should not encumber the actual programming too much. This last concern is the reason why arithmetical errors such as index out of range and division by zero are not prevented by most type systems.

The issue of readability wishes for the types to be succinct hints to the semantics of expressions. For example, when trying to understand the workings of a function, it is quite nice to know in advance that it computes booleans from lists of integers. This leads towards more expressive and intuitive types—limited by the need for decidability of type checking.

The ideal situation seems to arise when our type system admits automatic inference of types. Then we obtain all the advantages described above, without the need to bother with the type rules ourselves.

In the following sections we study the basic development of type systems for functional languages.

2 A Tiny Functional Language

We define a tiny vanilla-flavored functional language to serve as the basis for this presentation. It is designed to resemble the core of languages such as SCHEME, MIRANDA, or ML.

```
EXP ::= 0 | true | false | x |
        pred(EXP) | succ(EXP) | iszero(EXP) |
        cons(EXP,EXP) | car(EXP) | cdr(EXP) | nil | null(EXP) |
        if EXP then EXP else EXP fi |
        fun(x) EXP end | EXP(EXP) |
        let f = EXP in EXP end
```

Thus, we have integers, booleans, lists and functions. The let-construct defines a local scope and at the same time permits us to give simple recursive definitions.

The semantics of this language is entirely standard and will not be formalized. Note, though, that there are various expressions that we intuitively recognize as meaningless, such as: succ(true), false(0), and car(fun(x) x end). These are the bad expressions that our type system must exclude.

Exercise 2.1 *Program a* length *function on lists.* □

3 A Simple Type System

We choose a language of types that describes the different values of our expressions: integers, booleans, lists and functions.

$$\text{TYPE} ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{list}(\text{TYPE}) \mid \text{TYPE} \to \text{TYPE}$$

We must now decide on the rules for assigning types to expressions. Some are very easy; for example, true is of type Bool and fun(x) succ(x) end is of type Int → Int. Bad expressions are excluded by imposing type restrictions; for example, cdr only works on lists, the two branches of an if-expression must have the same type, and in function calls the formal and actual arguments types must be equal.

It would not be hard to write down a few manual pages describing this type system. However, we choose an extremely compact notation that is inspired by a connection between types and logic that we shall explore further in a later section. A type *judgment* is of the form:

$$A \vdash \mathsf{e} : \tau$$

Here A is a symbol table, i.e., a mapping from identifiers to types; e is an expression; and τ is a type. Its meaning is simply: relatively to the symbol table A, the expression e has type τ. Type judgments are combined into type *rules* of the form:

$$\frac{J_1;\ J_2;\ \ldots;\ J_n}{J}$$

Its meaning is: if the judgments J_i all hold, then so does the judgment J. Using these notational conventions, we can express all aspects of our type system in the following manner.

$$\frac{}{A \vdash \mathsf{0} : \mathsf{Int}} \qquad \frac{}{A \vdash \mathsf{true} : \mathsf{Bool}}$$

$$\frac{}{A \vdash \mathsf{false} : \mathsf{Bool}} \qquad \frac{A \vdash \mathsf{e} : \mathsf{Int}}{A \vdash \mathsf{pred(e)} : \mathsf{Int}}$$

$$\frac{A \vdash \mathsf{e} : \mathsf{Int}}{A \vdash \mathsf{succ(e)} : \mathsf{Int}} \qquad \frac{A \vdash \mathsf{e} : \mathsf{Int}}{A \vdash \mathsf{iszero(e)} : \mathsf{Bool}}$$

$$\frac{A \vdash \mathsf{e}_1 : \tau;\ A \vdash \mathsf{e}_2 : \mathsf{list}(\tau)}{A \vdash \mathsf{cons}(\mathsf{e}_1,\mathsf{e}_2) : \mathsf{list}(\tau)} \qquad \frac{A \vdash \mathsf{e} : \mathsf{list}(\tau)}{A \vdash \mathsf{car(e)} : \tau}$$

$$\frac{A \vdash \mathsf{e} : \mathsf{list}(\tau)}{A \vdash \mathsf{cdr(e)} : \mathsf{list}(\tau)} \qquad \frac{}{A \vdash \mathsf{nil} : \mathsf{list}(\tau)}$$

$$\frac{A \vdash \mathsf{e} : \mathsf{list}(\tau)}{A \vdash \mathsf{null(e)} : \mathsf{Bool}} \qquad \frac{A \vdash \mathsf{e}_1 : \mathsf{Bool};\ A \vdash \mathsf{e}_2 : \tau;\ A \vdash \mathsf{e}_3 : \tau}{A \vdash \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3\ \mathsf{fi} : \tau}$$

$$\frac{A,\mathsf{x} : \sigma \vdash \mathsf{e} : \tau}{A \vdash \mathsf{fun(x)\ e\ end} : \sigma \to \tau} \qquad \frac{A \vdash \mathsf{e}_1 : \sigma \to \tau;\ A \vdash \mathsf{e}_2 : \sigma}{A \vdash \mathsf{e}_1(\mathsf{e}_2) : \tau}$$

$$\frac{}{\ldots, \mathsf{x} : \sigma, \ldots \vdash \mathsf{x} : \sigma} \qquad \frac{A,\mathsf{f} : \sigma \vdash \mathsf{e}_1 : \sigma;\ A,\mathsf{f} : \sigma \vdash \mathsf{e}_2 : \tau}{A \vdash \mathsf{let}\ \mathsf{f} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2\ \mathsf{end} : \tau}$$

This is the Curry-Hindley type system dating back to the 1960s.

Exercise 3.1 *What is the rôle of the rule in the lower left-hand corner?* □

Exercise 3.2 *How can we tell that the* let*-construct allows recursive definitions?* □

The claim that a given expression has a particular type can be verified by a systematic (and seemingly pedantic) application of the above rules. Consider for example the function:

```
fun(g)
  fun(x)
    succ(g(x))
  end
end
```

We wish to show that it has type $(\mathsf{Bool} \to \mathsf{Int}) \to (\mathsf{Bool} \to \mathsf{Int})$. A formal derivation looks as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\mathsf{g} : \mathsf{Bool} \to \mathsf{Int}, \mathsf{x} : \mathsf{Bool} \vdash \mathsf{g} : \mathsf{Bool} \to \mathsf{Int}} \qquad \overline{\mathsf{g} : \mathsf{Bool} \to \mathsf{Int}, \mathsf{x} : \mathsf{Bool} \vdash \mathsf{x} : \mathsf{Bool}}}{\mathsf{g} : \mathsf{Bool} \to \mathsf{Int}, \mathsf{x} : \mathsf{Bool} \vdash \mathsf{g(x)} : \mathsf{Int}}}{\mathsf{g} : \mathsf{Bool} \to \mathsf{Int}, \mathsf{x} : \mathsf{Bool} \vdash \mathsf{succ(g(x))} : \mathsf{Int}}}{\mathsf{g} : \mathsf{Bool} \to \mathsf{Int} \vdash \mathsf{fun(x)\ succ(g(x))\ end} : \mathsf{Bool} \to \mathsf{Int}}}{\vdash \mathsf{fun(g)\ fun(x)\ succ(g(x))\ end\ end} : (\mathsf{Bool} \to \mathsf{Int}) \to (\mathsf{Bool} \to \mathsf{Int})}$$

Exercise 3.3 *Go through the above derivation bottom-up and explain which type rules are used.* □

It is quite easy to implement a top-down type checker based on such derivations, since the outermost type constructor and expression operator always determine the type rule that must be applied.

At this point we should pause to consider whether our type system is *sound*, in the sense that only good expressions are typable. To say anything about this, we would need a formal semantics of our language and a specification of goodness. The usual strategy is then to show a property called *subject reduction*: an expression of type τ can only evaluate to a value of type τ. Soundness then follows, since bad expressions do not evaluate to values of any type.

4 Simple Type Inference

As explained, we do not want to provide the types explicitly. Rather, we wish for the compiler to find appropriate typings of our expressions or to inform us if none exists. This spares us the trouble of writing down complicated type expression that often only clutter up our nice programs.

Another often cited advantage is the hope that a clever type inference algorithm may come up with more general types than we ourselves could, thus keeping the slack to an absolute minimum.

An obvious idea is to use a bottom-up procedure that traverses the parse tree and computes the type of every subexpression. But this only works for constant expressions without variables. As a trivial counter-example consider the term: succ(x). In a strict bottom-up procedure we must first choose a type for x. But unless we happen to select Int, we will mistakenly conclude that the expression is not typable.

We can still use a bottom-up procedure, but we should collect *all* types, rather than just a single specimen. However, we must somehow deal with the possibility that expressions may have infinitely many types. For example, the function:

```
fun(g)
   fun(x)
      succ(g(x))
   end
end
```

for which we earlier derived the type $(\mathsf{Bool} \to \mathsf{Int}) \to (\mathsf{Bool} \to \mathsf{Int})$, has in fact every possible type of the form $(\alpha \to \mathsf{Int}) \to (\alpha \to \mathsf{Int})$. But by this example we have already given the solution to our problem: an infinite set of types sometimes has a finite symbolic representation.

We define a *type scheme* to be a type that may contain occurrences of type variables. Formally, the syntax is as follows.

$$\begin{array}{lcl} \textsc{Types} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{list}(\textsc{Types}) \mid \textsc{Types}{\to}\textsc{Types} \mid \textsc{Var} \\ \textsc{Var} & ::= & \alpha \mid \beta \mid \ldots \end{array}$$

A type scheme defines a set of types: those that can be obtained by substituting types for variables in a consistent manner. For example, the type scheme $(\alpha \rightarrow \beta) \rightarrow \mathsf{list}(\alpha)$ defines the following infinite set:

$$\begin{array}{rcl}
(\mathsf{Int}\rightarrow\mathsf{Int}) & \rightarrow & \mathsf{list}(\mathsf{Int}) \\
(\mathsf{Bool}\rightarrow\mathsf{Int}) & \rightarrow & \mathsf{list}(\mathsf{Bool}) \\
(\mathsf{Bool}\rightarrow\mathsf{Bool}) & \rightarrow & \mathsf{list}(\mathsf{Bool}) \\
(\mathsf{Int}\rightarrow\mathsf{Bool}) & \rightarrow & \mathsf{list}(\mathsf{Int}) \\
 & \vdots & \\
((\mathsf{Int}\rightarrow\mathsf{Bool})\rightarrow\mathsf{list}(\mathsf{Bool})) & \rightarrow & \mathsf{list}(\mathsf{Int}\rightarrow\mathsf{Bool}) \\
 & \vdots &
\end{array}$$

The only requirement is that both occurrences of α must be substituted with the same type. If we also allow variables to be substituted with other type schemes, then we obtain a preorder $\sigma \geq \tau$ on type schemes, which holds exactly when τ is obtained from σ through a substitution.

Exercise 4.1 *What is a preorder?* □

Exercise 4.2 *Which pairs among the following type schemes are ordered?*

$$\begin{array}{l}
\alpha \\
\beta \\
\alpha \rightarrow \beta \\
\alpha \rightarrow \alpha \\
(\alpha \rightarrow \beta) \rightarrow \mathsf{list}(\alpha)
\end{array}$$

□

Exercise 4.3 *Argue that when $\sigma \geq \tau$, then the set of types determined by σ is a superset of that determined by τ.* □

The key to an efficient type inference algorithm is the following fundamental observation: for every typable expression e there is a type scheme σ that exactly defines all possible types of e. We call σ the *principal type scheme* for e. The task of type inference is then to construct the principal type scheme (for every subexpression) or to fail if the expression is untypable.

We present the algorithm in the style of Wand [11] from 1987. For every parse tree node n we introduce a type variable $[\![n]\!]$ which denotes the (as yet) unknown type scheme for the subexpression rooted in n. To capture scope rules correctly, we require that all parse tree nodes corresponding to occurrences of the same identifier in a scope must share the same type variable. In examples below, we use the slightly ambiguous notation $[\![\mathsf{e}]\!]$

for the type variable corresponding to the subexpression e.

Exercise 4.4 *Why is this notation ambiguous?* □

Every kind of expression imposes its own kind of constraints on type variables. Consider the case of a function application $\mathsf{e}_1(\mathsf{e}_2)$. We know from the type rule for application that the following equation must hold.

$$[\![\mathsf{e}_1]\!] = [\![\mathsf{e}_2]\!] \to [\![\mathsf{e}_1(\mathsf{e}_2)]\!]$$

That is, e_1 must be a function that accepts arguments of the same type as e_2. The full collection of constraints is as follows.

$$\begin{array}{rl}
\mathsf{0}: & [\![\mathsf{0}]\!] = \mathsf{Int} \\
\mathsf{true}: & [\![\mathsf{true}]\!] = \mathsf{Bool} \\
\mathsf{false}: & [\![\mathsf{false}]\!] = \mathsf{Bool} \\
\mathsf{pred(e)}: & [\![\mathsf{pred(e)}]\!] = [\![\mathsf{e}]\!] = \mathsf{Int} \\
\mathsf{succ(e)}: & [\![\mathsf{succ(e)}]\!] = [\![\mathsf{e}]\!] = \mathsf{Int} \\
\mathsf{iszero(e)}: & [\![\mathsf{iszero(e)}]\!] = \mathsf{Bool}, [\![\mathsf{e}]\!] = \mathsf{Int} \\
\mathsf{cons(e_1,e_2)}: & [\![\mathsf{cons(e_1,e_2)}]\!] = [\![\mathsf{e_2}]\!] = \mathsf{list}([\![\mathsf{e_1}]\!]) \\
\mathsf{car(e)}: & [\![\mathsf{e}]\!] = \mathsf{list}([\![\mathsf{car(e)}]\!]) \\
\mathsf{cdr(e)}: & [\![\mathsf{cdr(e)}]\!] = [\![\mathsf{e}]\!] = \mathsf{list}(\alpha) \\
\mathsf{nil}: & [\![\mathsf{nil}]\!] = \mathsf{list}(\alpha) \\
\mathsf{null(e)}: & [\![\mathsf{null(e)}]\!] = \mathsf{Bool}, [\![\mathsf{e}]\!] = \mathsf{list}(\alpha) \\
\mathsf{if\ e_1\ then\ e_2\ else\ e_3\ fi}: & [\![\mathsf{if\ e_1\ then\ e_2\ else\ e_3\ fi}]\!] = [\![\mathsf{e_2}]\!] = [\![\mathsf{e_3}]\!], [\![\mathsf{e_1}]\!] = \mathsf{Bool} \\
\mathsf{fun(x)\ e\ end}: & [\![\mathsf{fun(x)\ e\ end}]\!] = [\![\mathsf{x}]\!] \to [\![\mathsf{e}]\!] \\
\mathsf{e_1(e_2)}: & [\![\mathsf{e_1}]\!] = [\![\mathsf{e_2}]\!] \to [\![\mathsf{e_1(e_2)}]\!] \\
\mathsf{let\ f = e_1\ in\ e_2\ end}: & [\![\mathsf{let\ f = e_1\ in\ e_2\ end}]\!] = [\![\mathsf{e_2}]\!], [\![\mathsf{f}]\!] = [\![\mathsf{e_1}]\!]
\end{array}$$

In the above, we assume that each α is a fresh variable distinct from all others. We now know how to generate the constraints for any expression. For example, the expression:

```
fun(g)
   fun(x)
      succ(g(x))
   end
end
```

yields the following constraints.

$$\begin{aligned}
[\![\mathsf{fun(g)\ fun(x)\ succ(g(x))\ end\ end}]\!] &= [\![\mathsf{g}]\!] \to [\![\mathsf{fun(x)\ succ(g(x))\ end}]\!] \\
[\![\mathsf{fun(x)\ succ(g(x))\ end}]\!] &= [\![\mathsf{x}]\!] \to [\![\mathsf{succ(g(x))}]\!] \\
[\![\mathsf{succ(g(x))}]\!] &= [\![\mathsf{g(x)}]\!] = \mathsf{Int} \\
[\![\mathsf{g}]\!] &= [\![\mathsf{x}]\!] \to [\![\mathsf{g(x)}]\!]
\end{aligned}$$

Exercise 4.5 *Explain the origin of each of the above constraints.* □

We are then left with the problem of solving these constraints. We must assign to each type variable a type scheme such that all constraints are satisfied. Fortunately, this is exactly the well-known *unification problem*. It has an efficient linear-time algorithm that even computes a solution consisting of unique principal type schemes. The above example has the following solution.

$$\begin{aligned}
[\![\mathsf{fun(g)\ fun(x)\ succ(g(x))\ end\ end}]\!] &= ([\![\mathsf{x}]\!] \to \mathsf{Int}) \to ([\![\mathsf{x}]\!] \to \mathsf{Int}) \\
[\![\mathsf{fun(x)\ succ(g(x))\ end}]\!] &= [\![\mathsf{x}]\!] \to \mathsf{Int} \\
[\![\mathsf{succ(g(x))}]\!] &= \mathsf{Int} \\
[\![\mathsf{g(x)}]\!] &= \mathsf{Int} \\
[\![\mathsf{g}]\!] &= [\![\mathsf{x}]\!] \to \mathsf{Int} \\
[\![\mathsf{x}]\!] &= [\![\mathsf{x}]\!]
\end{aligned}$$

Exercise 4.6 *Verify that the above is indeed a solution.* □

We now present a simple algorithm for solving constraints. The constraint system is given as $\mathsf{x}_i = \sigma_i$. The solution S is a function from type variables to type schemes. We use the notation $S(\sigma)$ to indicate the type scheme obtained from σ by substituting every type variable with its value in S. If σ and τ are type schemes, then $\Delta(\sigma,\tau)$ is the set of two subtrees on which they first differ in a preorder traversal.

```
let $S(\mathsf{x}_i) = \mathsf{x}_i$;
while $S$ is not a solution do
    let $\{\mathsf{s},\mathsf{t}\} = \Delta(\mathsf{S}(\mathsf{x}_i),\mathsf{S}(\sigma_i))$, where $\mathsf{S}(\mathsf{x}_i) \neq \mathsf{S}(\sigma_i)$;
    if s and t are both type constructors then fail fi;
    Assume $\mathsf{s}=\mathsf{x}_j$;
    if $\mathsf{x}_j$ occurs in t then fail fi;
    let $S(\mathsf{x}_i) = \{\mathsf{x}_j \mapsto \mathsf{t}\}(S(\mathsf{x}_i))$
end
```

This algorithm, which was presented by Robinson [10] in 1965, runs in exponential time. The linear time algorithm is from 1978 by Paterson and Wegman [9].

Exercise 4.7 *Generate and solve the type constraints for the following expression.*

```
fun(x)
   fun(y)
      x((car(y))(x))
   end
end
```

□

When an expression is untypable, then we obtain an unsolvable collection of constraints. Consider the suspect expression: cons(succ(x),car(x)). It generates these constraints.

$$\begin{aligned}
[\![\mathsf{cons(succ(x),car(x))}]\!] &= \mathsf{list}([\![\mathsf{succ(x)}]\!]) \\
[\![\mathsf{car(x)}]\!] &= \mathsf{list}([\![\mathsf{succ(x)}]\!]) \\
[\![\mathsf{x}]\!] &= \mathsf{list}([\![\mathsf{car(x)}]\!]) \\
[\![\mathsf{succ(x)}]\!] &= \mathsf{Int} \\
[\![\mathsf{x}]\!] &= \mathsf{Int}
\end{aligned}$$

Exercise 4.8 *Argue that the above constraints are unsolvable.* □

5 Types and Logic

Earlier, the peculiar notation for type rules was blamed on their connection with *logic*. For very pure type systems, this connection is known as the *Curry-Howard isomorphism*. It is so famous that we shall frame it.

$$\boxed{\boxed{\textit{Types are formulas, and expressions are proofs.}}}$$

The isomorphism between types and formulas is simply syntactic: Int and Bool correspond to *facts*, and the type constructor $\rightarrow$ corresponds to the logical connective $\Rightarrow$. The relationship between expressions and proofs is a bit more subtle, but we only need to know its main consequence: a formula is valid if and only if the corresponding type is not empty.

Furthermore, our type rules mirror exactly the inference rules from logic. For example, there are two rules involving the function type:

$$\frac{A,\mathsf{x} : \sigma \vdash \mathsf{e} : \tau}{A \vdash \mathsf{fun(x)\ e\ end} : \sigma \to \tau} \qquad \frac{A \vdash \mathsf{e}_1 : \sigma \to \tau;\ A \vdash \mathsf{e}_2 : \sigma}{A \vdash \mathsf{e}_1(\mathsf{e}_2) : \tau}$$

If we ignore the expressions and apply the isomorphism, then we immediately recognize two familiar rules from logic:

$$\frac{A,\sigma \vdash \tau}{A \vdash \sigma \Rightarrow \tau} \qquad \frac{A \vdash \sigma \Rightarrow \tau;\ A \vdash \sigma}{A \vdash \tau}$$

These are respectively *deduction* and *modus ponens*. For example, consider the formula $(A \Rightarrow B) \Rightarrow ((C \Rightarrow A) \Rightarrow (C \Rightarrow B))$. Is this a tautology? Yes, because the expression:

```
fun(x)
   fun(y)
      fun(z)
         x(y(z))
      end
   end
end
```

has the principal type scheme $(\alpha \to \beta) \to ((\gamma \to \alpha) \to (\gamma \to \beta))$.

Exercise 5.1 *Why must a type scheme inferred for an expression necessarily be a tautology?* □

Exercise 5.2 *Complete the proof that* $(A \Rightarrow B) \Rightarrow ((C \Rightarrow A) \Rightarrow (C \Rightarrow B))$ *is a tautology.* □

In contrast, common sense tells us that the formula $(A \Rightarrow B) \Rightarrow (B \Rightarrow A)$ is not a tautology, which means that no expression has principal type scheme $(\alpha \to \beta) \to (\beta \to \alpha)$.

Exercise 5.3 *Argue that no expression has the principal type scheme* $(\alpha \to \beta) \to (\beta \to \alpha)$. *Reconcile this with the fact that e.g. the expression:*

```
fun(x)
   fun(y)
      0
   end
end
```

has type (Int → Bool) → (Bool → Int). □

Notice that, from a logical perspective, type inference is a rather bizarre activity: we have a proof and are looking for the corresponding formula.

Our tiny functional language so far excludes product and sum types. Using the Curry-Howard isomorphism, we can easily figure out how to add them. The usual rules from logic for conjunction give us the rules for products:

$$\frac{A \vdash \mathsf{e}_1 : \tau_1;\ A \vdash \mathsf{e}_2 : \tau_2}{A \vdash \mathsf{pair}(\mathsf{e}_1,\mathsf{e}_2) : \tau_1 \times \tau_2} \qquad \frac{A \vdash \mathsf{e} : \tau_1 \times \tau_2}{A \vdash \mathsf{fst}(\mathsf{e}) : \tau_1;\ A \vdash \mathsf{snd}(\mathsf{e}) : \tau_2}$$

There are certainly no surprises here. Similarly, we have an isomorphism between sums and disjunction:

$$\frac{A \vdash \mathsf{e}_1 : \tau_1}{A \vdash \mathsf{left}(\mathsf{e}_1) : \tau_1 + \tau_2} \qquad \frac{A \vdash \mathsf{e}_2 : \tau_2}{A \vdash \mathsf{right}(\mathsf{e}_2) : \tau_1 + \tau_2}$$

$$\frac{A \vdash \mathsf{e} : \tau_1 + \tau_2;\ A \vdash \mathsf{e}_1 : \tau_1 \to \sigma;\ A \vdash \mathsf{e}_2 : \tau_2 \to \sigma}{A \vdash \mathsf{decide}(\mathsf{e},\mathsf{e}_1,\mathsf{e}_2) : \sigma}$$

This should be rather familiar from a programming language point of view. However, closer scrutiny reveals that we have obtained some unusual logical rules: a proof of a disjunction must necessarily disclose which disjunct is being proved. This is not true for standard logic, which allows proofs to be considerably more indirect. The somewhat weaker logic that corresponds exactly to type theory is known as *intuitionistic logic* and has in fact been proposed by logicians for purely philosophical reasons.

Exercise 5.4 *What are the type inference constraints for these new expressions?* □

Exercise 5.5 *Find a type whose translation into logic is a classical tautology but which do not have any expressions and thus does not yield an intuitionistic tautology.* □

It is harder to see how the type rules for lists resemble logic in any way. But this is only because we have chosen the wrong operators. In the following version we see clearly the logical origins of lists.

$$\frac{A \vdash \mathsf{e}_1 : \tau;\ A \vdash \mathsf{e}_2 : \mathsf{list}(\tau)}{A \vdash \mathsf{cons}(\mathsf{e}_1,\mathsf{e}_2) : \mathsf{list}(\tau)}$$

$$\frac{A \vdash \mathsf{base} : \sigma;\ A \vdash \mathsf{step} : \sigma \times \tau \to \sigma}{A \vdash \mathsf{induct}(\mathsf{base},\mathsf{step}) : \mathsf{list}(\tau) \to \sigma}$$

Thus, lists are all about *counting* and *induction*. The induct operator is vastly more general than car and cdr. For example, the length operator is easily defined as: induct(0,fun(x) succ(fst(x)) end).

Exercise 5.6 *Use* induct *to define the operators* car *and* cdr. □

Still, the practical advantage of the connection between types and logic is that we have a source of inspiration for missing type rules. Consider a novel type constructor stream(τ) which defines *infinite* lists of τ-values. Such streams are in a certain formal sense the *dual* concept of lists. Correspondingly, the dual concept of induction is called *coinduction*. Looking up its definition, we magically get the following type rules for streams.

$$\frac{A \vdash \mathsf{base} : \sigma;\ A \vdash \mathsf{step} : \sigma \to \sigma \times \tau}{A \vdash \mathsf{coinduct}(\mathsf{base},\mathsf{step}) : \mathsf{stream}(\tau)}$$

$$\frac{A \vdash \mathsf{e} : \mathsf{stream}(\tau)}{A \vdash \mathsf{car}(\mathsf{e}) : \tau;\ A \vdash \mathsf{cdr}(\mathsf{e}) : \mathsf{stream}(\tau)}$$

The coinduct operator constructs streams—not inductively, but coinductively. And we might as well use the term *cocounting* to describe the actions of car and cdr.

Similar (co)induction principles are defined for arbitrary (monotone) recursive types.

Exercise 5.7 *Use* coinduct *to construct the stream* (0,1,2,3,4,...). □

Exercise 5.8 *Define the type rules for finite and infinite binary trees.* □

6 Polymorphic Types

The expression below seems at a first glance quite reasonable, but it is not typable. It is in fact an example of slack in our type system.

```
let f = fun(x)
            cons(x,nil)
        end
in pair(f(0),f(true))
```

The problem is of course that we must assign a single type to the function f. We need instead the usual notion of polymorphism. For this purpose we extend our type schemes as follows.

$$\begin{array}{lcl} \textsc{Types} & ::= & \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{list}(\textsc{Types}) \mid \textsc{Types} \rightarrow \textsc{Types} \mid \textsc{Var} \\ \textsc{Var} & ::= & \alpha \mid \beta \mid \ldots \\ \textsc{Poly} & ::= & \forall\ \textsc{Var}.\textsc{Poly} \mid \textsc{Types} \end{array}$$

The polymorphic type $\forall\alpha.\alpha \rightarrow \mathsf{list}(\alpha)$ describes a function that will accept an argument of any type and yield as result a list of values of that given type. This is exactly the type we want for the function f above. We will only allow polymorphism in connection with let-definitions.

The use of a universal quantifier hints at another connection with logic. Indeed, the types rules for polymorphic functions are those from logic, except that the layer of syntax is rather thicker this time.

$$\frac{A,\mathsf{f} : \sigma \vdash \mathsf{e}_1 : \sigma;\ A,\mathsf{f} : \forall\bar{\alpha}.\sigma \vdash \mathsf{e}_2 : \tau}{A \vdash \mathsf{let\ f} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2\ \mathsf{end} : \tau} \quad \bar{\alpha} \notin A$$

$$\frac{A \vdash \mathsf{f} : \forall\bar{\alpha}.\sigma}{A \vdash \mathsf{f} : \tau} \quad \sigma \geq \tau$$

Here $\bar{\alpha}$ indicates several type variables. The requirement $\bar{\alpha} \notin A$ means that A must not contain any assumptions about $\bar{\alpha}$. The requirement $\sigma \geq \tau$ means that τ is a specialization of σ, as described in Section 4 (except that only variables from $\bar{\alpha}$ may be substituted). This type system was introduced by Milner [7] in 1978.

Exercise 6.1 *What is the connection with logic?* □

Here is a formal derivation showing that the earlier example expression is typable. Unfortunately it is so large that we must cut it into several pieces.

$$\dfrac{\dfrac{\boxed{\mathbf{A}}}{\mathsf{f}{:}\alpha{\to}\mathsf{list}(\alpha) \vdash \mathsf{fun(x)}\ldots\mathsf{end} : \alpha{\to}\mathsf{list}(\alpha)} \quad \dfrac{\boxed{\mathbf{B}}}{\mathsf{f}{:}\forall\alpha.\alpha{\to}\mathsf{list}(\alpha) \vdash \mathsf{pair}(\ldots) : \mathsf{list(Int)}{\times}\mathsf{list(Bool)}}}{\vdash \mathsf{let\ f = fun(x)\ cons(x,nil)\ end\ in\ pair(f(0),f(true))} : \mathsf{list(Int)} \times \mathsf{list(Bool)}}$$

The $\boxed{\mathbf{A}}$-piece is concerned with typing the function f.

$$\dfrac{\dfrac{}{\mathsf{f} : \alpha{\to}\mathsf{list}(\alpha), \mathsf{x} : \alpha \vdash \mathsf{x} : \alpha} \quad \dfrac{}{\mathsf{f} : \alpha{\to}\mathsf{list}(\alpha), \mathsf{x} : \alpha \vdash \mathsf{nil} : \mathsf{list}(\alpha)}}{\mathsf{f} : \alpha{\to}\mathsf{list}(\alpha), \mathsf{x} : \alpha \vdash \mathsf{cons(x,nil)} : \mathsf{list}(\alpha)}$$

The $\boxed{\mathbf{B}}$-piece is concerned with typing the pair-expression.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{f} : \forall\alpha.\alpha{\to}\mathsf{list}(\alpha) \vdash \mathsf{f} : \forall\alpha.\alpha{\to}\mathsf{list}(\alpha)}}{\mathsf{f} : \forall\alpha.\alpha{\to}\mathsf{list}(\alpha) \vdash \mathsf{f} : \mathsf{Int}{\to}\mathsf{list(Int)}} \quad \dfrac{}{\mathsf{f} : \forall\alpha.\alpha{\to}\mathsf{list}(\alpha) \vdash \mathsf{0{:}Int}}}{\mathsf{f} : \forall\alpha.\alpha{\to}\mathsf{list}(\alpha) \vdash \mathsf{f(0)} : \mathsf{list(Int)}} \quad \boxed{\mathbf{C}}$$

Exercise 6.2 *Complete the* $\boxed{\mathbf{C}}$*-piece and verify each step in the above derivation.* □

Why is the requirement $\bar{\alpha} \notin A$ necessary? If we remove it, the function:

```
fun(x)
    let f = x in f
end
```

can be shown to have type scheme $\alpha \to \mathsf{Int}$. This is manifestly false, since the function acts as the identity. The purported derivation is as follows.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{x} : \alpha, \mathsf{f} : \alpha \vdash \mathsf{x} : \alpha} \quad \dfrac{\dfrac{}{\mathsf{x} : \alpha, \mathsf{f} : \forall\alpha.\alpha \vdash \mathsf{f} : \forall\alpha.\alpha}}{\mathsf{x} : \alpha, \mathsf{f} : \forall\alpha.\alpha \vdash \mathsf{f} : \mathsf{Int}}}{\mathsf{x} : \alpha \vdash \mathsf{let\ f = x\ in\ f} : \mathsf{Int}}}{\vdash \mathsf{fun(x)\ let\ f = x\ in\ f\ end} : \alpha \to \mathsf{Int}}$$

Exercise 6.3 *Catch the error in the above derivation.* □

Exercise 6.4 *What is the corresponding logical fallacy?* □

7 Polymorphic Type Inference

The task of type inference seems to become vastly more complex with the introduction of the polymorphic let-construct. However, there is certainly a very naïve idea that we can fall back on. Using a simple syntactic transformation that unfolds let-definitions, we can expand a polymorphic program into an equivalent monomorphic version. For the expression:

```
let f = fun(x)
            cons(x,nil)
        end
in pair(f(0),f(true))
```

we obtain the equivalent version:

```
     ⎛ let f = fun(x)              let f = fun(x)              ⎞
     ⎜             cons(x,nil)                 cons(x,nil)     ⎟
pair ⎜         end              ,          end                 ⎟
     ⎝ in f(0)                     in f(true)                  ⎠
```

which certainly is typable in the monomorphic type system. The only disadvantage is that the expanded version of an expression may be exponentially larger than the original.

Exercise 7.1 *Find an example where a program of size $O(n)$ expands to one of size $\Omega(2^n)$.* □

Exercise 7.2 *Argue that the expanded version is typable in the monomorphic system if and only if the original version is typable in the polymorphic system.* □

Can we do better than this exponential algorithm? An obvious idea is to use a form of dynamic programming where the principal type schemes of let-definitions are saved, so that they need only be computed once. This is essentially the Damas-Milner [1] algorithm from 1982 which is implemented in the ML system. For almost ten years it was folklore that this algorithm had low polynomial time complexity. This also corresponded well with the practical experiences of ML programmers.

This belief was thorougly shattered when Kfoury, Tiuryn, and Urzyczyn [4] and Mairson [6] in 1989 simultaneously proved that the polymorphic type inference problem is complete for exponential time. This means that any correct implementation must use an exponential amount of time on infinitely many inputs. His proof is an awesome construction of an ML program that directly simulates a given deterministic Turing machine running in exponential time, and where typability of the program coincides with acceptance by the machine. PSPACE-hardness alone was proved by Kanellakis and Mitchell [3] earlier in 1989.

In spite of this nasty result, implementations of the ML language seem to be running very well in everyday life. But danger lurks beneath the surface. As a concrete example, regard the following ML program.

```
fun pair x y = fn z => z x y;
let val x1=fn y => pair y y in
  let val x2=fn y => x1(x1(y)) in
    let val x3=fn y => x2(x2(y)) in
      let val x4=fn y => x3(x3(y)) in
        x4(fn z=>z)
      end
    end
  end
end;
```

Its principal type scheme is:

```
(((((((((((((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  -> ’a) -> (’a -> ’a)
-> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c)
-> ’d) -> ’d) ->  (((((((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  ->  ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->
’c) -> ’c) -> ’d) -> ’d) -> ’e) -> ’e) ->  (((((((((’a  -> ’a) -> (’a ->
’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) ->
’c) -> (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a) -> (’a
-> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) ->  (((((((’a -> ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->
’c) -> ’c) -> (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a)
-> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> ’e) -> ’e) ->
’f) -> ’f) ->  (((((((((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  ->  ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->
’c) -> ’c) -> ’d) -> ’d) -> (((((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b)
-> (((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> (((((’a  ->
’a) -> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b) ->
’b) -> ’c) -> ’c) -> ’d) -> ’d) -> ’e) -> ’e) ->  (((((((((’a  -> ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->
’c) -> ’c) -> (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a)
-> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> (((((((’a  ->
’a) -> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b) ->
’b) -> ’c) -> ’c) -> (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> ’e) ->
’e) -> ’f) -> ’f) -> ’g) -> ’g) ->  (((((((((((((’a -> ’a) -> (’a -> ’a)
-> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c)
->  (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a) -> (’a ->
’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> (((((((’a -> ’a) -> (’a
-> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c)
-> ’c) -> (((((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) ->  ’e)  ->  ’e) ->
(((((((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a) -> (’a ->
’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  -> ’a) -> (’a -> ’a) -> ’b)
-> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d)
-> ’d) ->  (((((((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a)
-> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  ->  ’a) -> (’a ->
’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) ->
’c) -> ’d) -> ’d) -> ’e) -> ’e) -> ’f) -> ’f) -> (((((((((((’a -> ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->
’c) -> ’c) -> (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a)
-> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> (((((((’a  ->
’a) -> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b) ->
’b) -> ’c) -> ’c) -> (((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> ’e) ->
’e) -> (((((((((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a -> ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  -> ’a) -> (’a -> ’a)
-> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c)
-> ’d) -> ’d) ->  (((((((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) ->  (((((’a  ->  ’a) ->
(’a -> ’a) -> ’b) -> ’b) -> (((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->
’c) -> ’c) -> ’d) -> ’d) -> ’e) -> ’e) -> ’f) -> ’f) ->  ’g)  ->  ’g) ->
’h) -> ’h) ->  (((((((((((((((’a  -> ’a) -> (’a -> ’a) ->  ’b) -> ’b) ->
(((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> (((((’a -> ’a)
-> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b)
-> ’c) -> ’c) -> ’d) -> ’d) ->  (((((((’a -> ’a) -> (’a -> ’a) -> ’b) ->
’b) -> (((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> (((((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b)
-> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> ’e) -> ’e) ->  (((((((((’a  -> ’a)
-> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b) -> ’b)
-> ’c) -> ’c) -> (((((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> (((’a ->
’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) -> (((((((’a
-> ’a) -> (’a -> ’a) -> ’b) -> ’b) ->  (((’a -> ’a) -> (’a -> ’a) -> ’b)
-> ’b) -> ’c) -> ’c) ->  (((((’a  -> ’a) -> (’a -> ’a) ->  ’b) -> ’b) ->
(((’a  -> ’a) -> (’a -> ’a) -> ’b) -> ’b) -> ’c) -> ’c) -> ’d) -> ’d) ->
```

```
'e) -> 'e) -> 'f) -> 'f) -> ((((((((((('a -> 'a) -> ('a -> 'a) -> 'b) ->
'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> ((((('a
-> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b)
-> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> ((((((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) ->
((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a)
-> 'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> 'e) -> 'e) -> ((((((((('a
-> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b)
-> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) ->
((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) ->
((((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a ->
'a) -> 'b) -> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) -> ('a -> 'a) -> 'b)
-> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd)
-> 'd) -> 'e) -> 'e) -> 'f) -> 'f) -> 'g) -> 'g) -> ((((((((((((('a ->
'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) ->
'b) -> 'c) -> 'c) -> ((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a
-> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) ->
((((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a ->
'a) -> 'b) -> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) -> ('a -> 'a) -> 'b)
-> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd)
-> 'd) -> 'e) -> 'e) -> ((((((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b)
-> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> ((((('a ->
'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) ->
'b) -> 'c) -> 'c) -> 'd) -> 'd) -> ((((((('a -> 'a) -> ('a -> 'a) -> 'b)
-> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) ->
((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a)
-> 'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> 'e) -> 'e) -> 'f) -> 'f) ->
((((((((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a
-> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) ->
'd) -> 'd) -> ((((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a ->
'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) -> ('a
-> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c)
-> 'c) -> 'd) -> 'd) -> 'e) -> 'e) -> ((((((((('a -> 'a) -> ('a -> 'a)
-> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c)
-> ((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a ->
'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> ((((((('a -> 'a) -> ('a
-> 'a) -> 'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> 'c)
-> 'c) -> ((((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) ->
('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> 'e) -> 'e) ->
'f) -> 'f) -> 'g) -> 'g) -> 'h) -> 'h) -> 'i) -> 'i
```

Note that by the analogy described in Section 5, the above is a gigantic tautology in propositional logic.

8 Polymorphism and Recursion

We have only allowed polymorphism in let-definitions, and even there we have some limitations. For example, the following reasonable expression is not typable—we have some slack in the polymorphic type system.

```
let f = fun(i)
          fun(x)
            if iszero(i) then x
                         else f(pred(i))(pair(x,x))
            fi
          end
        end
in f(87)(true)
```

Exercise 8.1 *Argue that the above expression is not typable in the polymorphic system.* □

Exercise 8.2 *What is the result of the above expression?* □

There are also less contrived examples showing that this restriction is a practical nuisance. The general problem is that a function is only polymorphic from the outside, so to speak. However, this can be remedied by a minor change in our type rule.

$$\frac{A,\mathsf{f} : \forall\bar{\alpha}.\sigma \vdash \mathsf{e}_1 : \sigma;\ A,\mathsf{f} : \forall\bar{\alpha}.\sigma \vdash \mathsf{e}_2 : \tau}{A \vdash \mathsf{let}\ \mathsf{f} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2\ \mathsf{end} : \tau} \quad \bar{\alpha} \notin A$$

Exercise 8.3 *What has been changed in the type rule?* □

This is the Milner-Mycroft [8] type system from 1984. A corresponding type inference algorithm must necessarily be more obscure than the earlier one, since the naïve expansion of let-definitions may now yield infinite monomorphic versions. Yet, implementations did exist for versions of the ML language. Thus it caused some concern when Henglein [2] and Kfoury, Tiuryn, and Urzyczyn [5] in 1990 simultaneously proved that the type inference problem is in fact undecidable.

The problem seems to obey the laws of cartoon physics, since people were perfectly happy with the proposed implementations before the undecidability was known. The reason is that the semi-algorithm terminates for all typable expressions and only fails to terminate for a very small fraction of untypable expressions.

9 Higher Type Systems

By no means does the development of type systems stop here. There are at least three good reasons to push on.

Firstly, there is always more slack to pick up. Many interesting type rules have been motivated by annoying examples of unfairly rejected expressions. One example is the *conjunctive* type written $\vdash \mathsf{e} : \sigma \wedge \tau$, which means that e simultaneously has types σ and τ.

Exercise 9.1 *Suggest rules for conjunctive types. How do they differ from products?* □

Polymorphism may also be included as an orthogonal feature in the type system. Thus types may look like: $\forall\alpha.\alpha \rightarrow (\forall\beta.\beta \rightarrow \beta \rightarrow \beta) \rightarrow \alpha$ This is the system F2 studied by Girard and Reynolds. Such extensions are naturally beyond the reach of type inference.

Exercise 9.2 *Suggest an expression that could have the polymorphic type* $\forall\alpha.\alpha \rightarrow (\forall\beta.\beta \rightarrow \beta \rightarrow \beta) \rightarrow \alpha$. □

Secondly, we might stray from a purely functional language. Many efforts have been directed towards incorporating non-functional features into ML-style type systems. For example, the mixture of polymorphism and pointers is very complicated and often requires subtle type rules.

Exercise 9.3 *What is wrong with these simple rules for pointers (assume the obvious semantics for the expressions)?*

$$\frac{A \vdash \mathsf{e} : \tau}{A \vdash \mathsf{ref(e)} : \mathsf{pointer}(\tau)} \qquad \frac{A \vdash \mathsf{e}\text{:}\ \mathsf{pointer}(\tau)}{A \vdash \mathsf{deref(e)} : \tau}$$

$$\frac{A \vdash \mathsf{x} : \mathsf{pointer}(\tau);\ A \vdash \mathsf{e} : \mathsf{pointer}(\tau)}{A \vdash \mathsf{x{:=}e} : \mathsf{pointer}(\tau)}$$

□

Thirdly, it is possible to have types that capture more of the semantics of expressions. This is done by emphasizing the logical connection and building type systems so rich that one can define e.g. a type whose values are just the sorting functions on integer lists. At this point type checking is equivalent to program verification and, hence, undecidable. This means that each program must include a hand-written but compiler-checked proof of its type-correctness.

References

[1] Luis Damas and Robin Milner. Principal type schemes for functional programming. In *9th Symposium on Principles of Programming Languages*, 1982.

[2] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15, 1993.

[3] P. Kanellakis and J. Mitchell. Polymorphic unification and ML typing. In *16th Symposium on Principles of Programming Languages.* ACM Press, January 1989.

[4] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Ml typability is DEXPTIME-complete. In *15th Colloquium on Trees in Algebra and Programming.* Springer-Verlag, May 1990.

[5] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15, 1993.

[6] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *17th Symposium on Principles of Programming Languages.* ACM Press, January 1990.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[8] A. Mycroft. Polymorphic type schemes and recursive definitions. In *6th International Conference on Programming.* Springer-Verlag, 1984.

[9] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16, 1978.

[10] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 1965.

[11] M. Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X, 1987.

10 Problems

Problem 10.1 Give a formal derivation showing that the expression:

```
let Double = fun(x)
                 if iszero(x) then 0
                                  else succ(succ(Double(pred(x))))
                 fi
               end
in Double(succ(0))
```

has type Int.

Problem 10.2 Infer the principal type scheme for this expression:

```
fun(x)
   fun(y)
      cons(car(x),y(true))
   end
end
```

Problem 10.3 List all constant expressions having these polymorphic types.

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

Problem 10.4 Oyster of the week. Here are the rules for the existential quantifier in logic.

$$\frac{A \vdash \sigma[\alpha \leftarrow \tau]}{A \vdash \exists\alpha.\sigma} \qquad \frac{A \vdash \exists\alpha.\sigma;\ A,\sigma \vdash \tau}{A \vdash \tau}\ \alpha \notin A, \alpha \notin \tau$$

The notation $\sigma[\alpha \leftarrow \tau]$ means σ with all occurrences of α substituted with τ. Explain the logical content of these rules. Introduce an existential quantifier into our type system, along with a reasonable set of expressions, and recognize an important concept from programming languages.

A Selection of Solutions

Exercise 3.1 It allows the type checker to perform lookups in the symbol table. □

Exercise 3.2 In the judgment for the expression being let-defined, we include f in the symbol table: $A,\mathsf{f} : \sigma \vdash \mathsf{e}_1 : \sigma$. □

Exercise 4.1 It is a relation that is reflexive and transitive, but fails to be anti-symmetric, e.g. $\alpha \leq \beta$ and $\beta \leq \alpha$ but $\alpha \neq \beta$. □

Exercise 4.2

$$\begin{array}{rcl} \alpha & \geq & \{\alpha, \beta, \alpha \to \beta, \alpha \to \alpha, (\alpha \to \beta) \to \mathsf{list}(\alpha)\} \\ \beta & \geq & \{\alpha, \beta, \alpha \to \beta, \alpha \to \alpha, (\alpha \to \beta) \to \mathsf{list}(\alpha)\} \\ \alpha \to \beta & \geq & \{\alpha \to \beta, \alpha \to \alpha, (\alpha \to \beta) \to \mathsf{list}(\alpha)\} \end{array}$$

□

Exercise 4.3 Simply compose the substitutions. □

Exercise 4.4 Two subexpressions with identical syntax may yield different type variables, such as e.g. $[\![\mathsf{nil}]\!]$. However, from the context of each type constraint it is clear to which subexpression a type variable corresponds. □

Exercise 4.8 The two constraints $[\![\mathsf{x}]\!] = \mathsf{Int}$ and $[\![\mathsf{x}]\!] = \mathsf{list}([\![\mathsf{car(x)}]\!])$ are contradictory. □

Exercise 5.1 The rules for inferring types correspond to valid inference rules of logic. □

Exercise 5.3 A function with type scheme $(\alpha \to \beta) \to (\beta \to \alpha)$ must look like:

```
fun(x)
  fun(y)
    e
  end
end
```

where e has type α. An exhaustive search through the type rules convinces us that we cannot find such an expression. The fact that an expression has type $(\mathsf{Int} \to \mathsf{Bool}) \to (\mathsf{Bool} \to \mathsf{Int})$ is not a problem, since this is merely *one* instance of the type scheme. □

Exercise 5.4

$$\begin{array}{rl}
\mathsf{pair(e_1,e_2)}: & [\![\mathsf{pair(e_1,e_2)}]\!] = [\![\mathsf{e_1}]\!] \times [\![\mathsf{e_2}]\!] \\
\mathsf{fst(e)}: & [\![\mathsf{e}]\!] = [\![\mathsf{fst(e)}]\!] \times \alpha \\
\mathsf{snd(e)}: & [\![\mathsf{e}]\!] = \alpha \times [\![\mathsf{snd(e)}]\!] \\
\mathsf{left(e_1)}: & [\![\mathsf{left(e_1)}]\!] = [\![\mathsf{e_1}]\!] + \alpha \\
\mathsf{right(e_2)}: & [\![\mathsf{right(e_2)}]\!] = \alpha + [\![\mathsf{e_2}]\!] \\
\mathsf{decide(e,e_1,e_2)}: & [\![\mathsf{e}_i]\!] = [\![\mathsf{e}]\!] \rightarrow [\![\mathsf{decide(e,e_1,e_2)}]\!] \\
 & [\![\mathsf{e}]\!] = \alpha + \beta \\
 & [\![\mathsf{e_1}]\!] = \alpha \rightarrow [\![\mathsf{decide(e,e_1,e_2)}]\!] \\
 & [\![\mathsf{e_2}]\!] = \beta \rightarrow [\![\mathsf{decide(e,e_1,e_2)}]\!]
\end{array}$$

□

Exercise 5.5 The formula $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ is a classical tautology but translates into an empty type. □

Exercise 5.6 We can only solve this problem, if we allow an expression error that generates a run-time error. Then we have

$$\begin{array}{l}
\mathsf{car = induct(error,fun(x)\ snd(x)\ end)} \\
\mathsf{cdr = induct(error,fun(x)\ fst(x)\ end)}
\end{array}$$

□

Exercise 5.7 (0,1,2,3,4,...) = coinduct(0,fun(x) pair(succ(x),x) end. □

Exercise 5.8 Let finbin(τ) be the type of finite binary trees with τ-values at all nodes. We then have:

$$\frac{A \vdash \mathsf{e} : \tau;\ A \vdash \mathsf{e_1} : \mathsf{finbin}(\tau);\ A \vdash \mathsf{e_2} : \mathsf{finbin}(\tau)}{A \vdash \mathsf{bin(e,e_1,e_2)} : \mathsf{finbin}(\tau)}$$

$$\frac{A \vdash \mathsf{base} : \sigma;\ \mathsf{step} : \sigma \times \tau \times \sigma \rightarrow \sigma}{A \vdash \mathsf{induct(base,step)} : \mathsf{finbin}(\tau) \rightarrow \sigma}$$

Let infbin(τ) be the type of infinite binary trees with τ-values at all nodes. We then have:

$$\frac{A \vdash \mathsf{base} : \sigma;\ \mathsf{step} : \sigma \rightarrow \sigma \times \tau \times \sigma}{A \vdash \mathsf{coinduct(base,step)} : \mathsf{infbin}(\tau)}$$

$$\frac{A \vdash \mathsf{e} : \mathsf{infbin}(\tau)}{A \vdash \mathsf{root(e)} : \tau;\ A \vdash \mathsf{leftson(e)} : \mathsf{infbin}(\tau);\ A \vdash \mathsf{rightson(e)} : \mathsf{infbin}(\tau)}$$

□

Exercise 6.1 The logical rules for universal quantification are:

$$\frac{A \vdash \sigma}{A \vdash \forall\alpha.\sigma}\ \alpha \notin A \qquad \frac{A \vdash \forall\alpha.\sigma}{A \vdash \sigma[\alpha \leftarrow \tau]}$$

□

Exercise 6.3 It is not legal to make f polymorphic in α since we have the assumption x : α. □

Exercise 7.1 The ML program on page 17 is like this (with $n = 4$). □

Exercise 7.2 In both approaches we are allowed to choose fresh type variables for the types of the let-defined expressions. □

Exercise 8.1 The type variable $[\![\mathsf{x}]\!]$ must satisfy $[\![\mathsf{x}]\!] = [\![\mathsf{x}]\!] \times [\![\mathsf{x}]\!]$, which is not possible. □

Exercise 8.2 It computes a complete binary tree of height 87 with the value true at each leaf. □

Exercise 8.3 We have universally quantified the type of f in the judgment of e_1. □

Exercise 9.3 The type rules allow the typing of bad expressions, such as:

```
let r = ref(fun(x) x end)
in
let x = r:=ref(fun(x) succ(x) end)
in
deref(r)(true)
```

This is type correct, since r is given type $\forall\alpha.$pointer($\alpha \rightarrow \alpha$) and thus can be instantiated to both pointer(Int→Int) and pointer(Bool→Bool). However, its computation leads to succ(true). □

Problem 10.3 There are no constant expressions of type $\forall\alpha.\alpha$; for $\forall\alpha.$list(α) there is only nil; for $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$ we have two functions: fun(x) fun(y) x end end and fun(x) fun(y) y end end; finally, for $\forall\alpha.\forall\beta.(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta$ there are infinitely many functions of the form: fun(f) fun(g) fun(x) f(g^i(x)) end end end for $i > 0$. □

Problem 10.4 If we can prove σ with any specific value τ in place of the variable α, then τ is a witness to the truth of $\exists\alpha.\sigma$. Conversely, if this existential quantification holds and we can prove τ using σ but without any assumptions on α, then we can conclude τ.

This corresponds to abstract data types, where the implementation type is hidden. We only need to know that is exists! The type rules are as follows:

$$\frac{A \vdash \mathsf{e} : \sigma[\alpha \leftarrow \tau]}{A \vdash \mathsf{abs(e)} : \exists\alpha.\sigma} \quad \frac{A \vdash \mathsf{e}_1 : \exists\alpha.\sigma;\ A,\mathsf{x} : \sigma \vdash \mathsf{e}_2 : \tau}{A \vdash \mathsf{use\ x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 : \tau}\ \alpha \notin A, \alpha \notin \tau$$

The side conditions exactly state that the implementation type is hidden during use. The type of an abstract implementation of a stack of booleans could be:

$$\exists\alpha.(\mathsf{Int} \to \alpha) \times (\alpha \to \mathsf{Bool}) \times (\alpha \times \mathsf{Bool} \to \alpha) \times (\alpha \to \alpha \times \mathsf{Bool})$$

corresponding to the stack operations Init, Empty, Push, and Pop. In a concrete implementation the hidden type α could be list(Bool).

A polymorphic, abstract stack would then have the type:

$$\forall\beta.\exists\alpha.(\mathsf{Int} \to \alpha) \times (\alpha \to \mathsf{Bool}) \times (\alpha \times \beta \to \alpha) \times (\alpha \to \alpha \times \beta)$$

□