A Fatal Weakness in Simple Structural Typing

Structural similar types like `list-of-int` and `list-of-bool` are completely separate. Standard list operations that do not depend on the element type must be rewritten for every different element type. There are no common abstractions connecting `list-of-int` and `list-of-bool` because they are completely disjoint types akin to `int` and `bool`.

The solution is to introduce type parameterization (polymorphism) into the data domain and the corresponding type system. Instead of defining

```plaintext
int-list :: = empty() | cons(int, int-list)  
bool-list :: = empty() | cons(bool, bool-list)  
...  
```

we define a single parameterized form of list:

```plaintext
list T :: = empty() | cons(T, list T)  
```
What Types Correspond to Parametric Data?

In the data definition:

\[
\text{list } T :: = \text{ empty() } \mid \text{ cons}(T, \text{ list } T)
\]

what are the types of data operations like \text{empty}, \text{cons}, and the ... corresponding accessors? We need to introduce the notion of type schemes. A type scheme has syntax

\[
\forall \alpha_1 \cdot \cdot \cdot \alpha_n . \tau
\]

where \(\alpha_1, \ldots, \alpha_n\) are type variables, and \(\tau\) is a conventional type that may be expressed in terms of \(\alpha_1, \ldots, \alpha_n\) and perhaps other type variables (constrained in the relevant type environment). The types for the data operations in our example are:

- \text{empty}: \forall \alpha \ (\rightarrow \text{ list } \alpha)
- \text{cons}: \forall \alpha \ (\alpha \times \text{ list } \alpha \rightarrow \text{ list } \alpha)
- \text{cons-1}: \forall \alpha \ (\text{ list } \alpha \rightarrow \alpha)
- \text{cons-2}: \forall \alpha \ (\text{ list } \alpha \rightarrow \text{ list } \alpha)
How Are Type Schemes Used in Inference?

Two Options:

I. First option: explicit polymorphism. We add explicit type abstraction and application to the programming language.

\[
M ::= \lambda V: \tau. M \mid (M M) \mid V \mid \Lambda T. M \mid (M \tau)
\]

\[
\tau ::= D \mid (\tau \rightarrow \tau) \mid \forall T \tau
\]

where \( V \) is the set of vars and \( T \) is the set of type vars. The symbol \( \Lambda \) is a capital \( \lambda \); it denotes type abstraction.

Typing rules:

Fun abstraction, application as before

\[
\Gamma \vdash M : \tau, \quad \alpha \text{ not free in } \Gamma
\]

\[
\Gamma \vdash \Lambda \alpha. M : \forall \alpha \tau
\]

\[
\Gamma \vdash (M \sigma) : \tau_{[\alpha := \sigma]}
\]

Called the polymorphic \( \lambda \)-calculus or System F. Clumsy in practice. Influenced Java 5 type system.
Implicit Polymorphism

II. Second option for interpreting type schemes.

(i) We restrict the body of a type scheme to an ordinary (non-schematic) type. Hence, $\forall$ can only appear at the top-level in a type. We implicitly close top-level types.

(ii) We make no changes to the programming language, which looks like an extension of the untyped lambda-calculus.

Typing rules same as ext typed lambda calculus, except

$$
\Gamma, x : \sigma \vdash M : \tau
$$

$$
\frac{\Gamma \vdash \lambda x. M : \sigma \rightarrow \tau}{\Gamma \vdash \lambda x. M : \sigma \rightarrow \tau}
$$

Extra axiom: $\Gamma, \{x : \forall T S\} \vdash x : S'$

where $S'$ is any substitution instance of $S$ (replacing $T$).
Implicit Polymorphism cont.

Different instantiations of same type scheme axiom may use different terms in the substitution for the type variable:
\[ \Gamma, \{ x : \forall T(T \rightarrow T) \} \vdash x : \text{int} \rightarrow \text{int} \]
\[ \Gamma, \{ x : \forall T(T \rightarrow T) \} \vdash x : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \]

The preceding system enables us to use primitive operations with schematic types because the types of primitive operations are built into the base environment, but how do we define new polymorphic operations? We need to revise our language so that let and letrec introduce polymorphic operations!
Defining Polymorphic Functions

The following polymorphic let construct was Milner's greatest insight in devising ML. Consider the Jam program

\[
\text{let } \text{id} := \text{map } x \rightarrow x; \text{ in } (\text{id}(\text{id}))(4)
\]

If we interpret `let` as before, this program is untypable because `id` is used two different ways: as the identity function for type `int \rightarrow int` and for type `int`.

But we can revise (liberalize) our typing rule for `let`

\[
\frac{\Gamma \mid M : \sigma; \Gamma, x : \text{close}(\Gamma, \sigma) \mid N : \tau}{\Gamma \mid \text{let } x := M \text{ in } N : \tau} \quad \text{(non-rec let poly rule)}
\]

where `\text{close}(\Gamma, \sigma)` means find all of the free type variables \(\alpha_1, \ldots, \alpha_n\) in \(\sigma\) that do not appear in \(\Gamma\) and generate \(\forall \alpha_1, \ldots, \alpha_n \sigma\).
Type Reconstruction

Implicit polymorphism is far more important in practice than explicit polymorphism because the types in implicitly typed program can easily be reconstructed if they are erased. (This process is often called “type inference” but we will use the term “reconstruction” instead of “inference” because we want to use the term “inference” to refer to formally proving programs are typable using typing rules.)

How does type reconstruction work? Build the type inference tree for a program using the typing rules with type variables for the types of all lambda variables. To make this tree a valid proof tree, certain equality relationships must hold between type expressions (these equality constraints appear in the statement of the rules). Generate the list of equality constraints and solve them using unification. Unification always determines the most “general” solution (a substitution) to the set of symbolic equality constraints on type terms for program expressions in the sense that all possible solutions are substitution instances of the most general one.

This reconstruction process is algorithmic! Moreover, it always the most general reconstruction possible.
Example

Let’s perform type reconstruction for the program

```plaintext
let id := map x to x; in (id(id))(4)
```

using the algorithm that we just described. We build the type inference tree for program in the obvious way (using the rule for the outermost operation in each internal node of the type inference tree starting with our program at the root).

Recall that our expanded language (beyond the Simply Typed Lambda Calculus, slightly smaller than LC, that was our first typed language), we need a base type assignment (environment mapping type variables to types) \( \Gamma_0 \) that assigns types to all of the constants (values including primitive operations). (Note that \( \Gamma_0 \) is an infinite table since we have infinitely many constants [including all integers].) So the root of inference tree is

```
\Gamma_0 \vdash let id := map x to x; in (id(id))(4): \alpha
```

with outermost construct recursive `let`. The choice of the type variable \( \alpha \) is arbitrary. Recall that typing rule recursive `let` is almost identical to the typing rule for ordinary `let`, and that the rule for ordinary `let` can be derived by expanding `let` into the application of a `map` to the right hand sides (each is an argument) of the bindings introduced by the `let`. Hence, the premises of the rule look like the premises of an ordinary application except for the expanded type environment (to support recursion!) for typing the right hand sides of the bindings.

Note that the CBV/CBN and lazy/eager distinctions in interpretation have no effect on our typing rules.
Example

Let $\Gamma_1 = \Gamma_0, \text{id}: \forall \alpha (\alpha \to \alpha)$. Then

$$\Gamma_1 \vdash \text{id}:(\text{int} \to \text{int}) \to (\text{int} \to \text{int}), \ \Gamma_1 \vdash \text{id} : \text{int} \to \text{int}$$

\[ \begin{align*}
\Gamma_0, x : \alpha & \vdash x : \alpha & \Gamma_1 & \vdash \text{id(id)} : \text{int} \to \text{int}, \ \Gamma_1 & \vdash 4 : \text{int} \\
\hline
\Gamma_0 & \vdash \text{map} \ x \ \text{to} \ x : \alpha \to \alpha, & \Gamma_1 & \vdash (\text{id(id)})(4) : \text{int} \\
\hline
\Gamma_0 & \vdash \text{let} \ \text{id} := \text{map} \ x \ \text{to} \ x; \ \text{in} \ (\text{id(id)})(4) : \text{int}
\end{align*} \]