Comp 411
Principles of Programming Languages
Lecture 20
Polymorphic Types

Corky Cartwright

October 31, 2012

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.

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

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

we define a single parameterized form of list:

```
list T :: = unit() | cons(T, list T)
```

What Types Correspond to Parametric Data?

In the data definition:

```
list T :: = unit() | cons(T, list T)
```

what are the types of data operations like `unit`, `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 α_1, ..., α_n are type variables, and τ is a conventional type. The types of the data operations in our example are:

```
unit: ∀α ( → list α)
cons: ∀α (α x list α → list α)
cons-1: ∀α (list α → α)
cons-2: ∀α (list α → list α)
```

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.

$$\mathbf{M} ::= \lambda\, \mathbf{V}{:}\tau\,.\ \mathbf{M} \mid (\mathbf{M}\ \mathbf{M}) \mid \mathbf{V} \mid \Lambda\, \mathbf{T}.\ \mathbf{M} \mid (\mathbf{M}\ \tau)$$

$$\tau ::= \mathbf{D} \mid (\tau \to \tau) \mid \forall\, \mathbf{T}\, \tau$$

where V is the set of vars and T is the set of type vars

Typing rules:

Fun abstraction, application as before

$$\frac{\Gamma \mid \mathbf{M}{:}\ \tau, \quad \alpha \text{ not free in } \Gamma}{\Gamma \mid \Lambda\ \alpha.\mathbf{M}{:}\ \forall\alpha\,\tau} \qquad \frac{\Gamma \mid \mathbf{M}{:}\ \forall\alpha\,\tau}{\Gamma \mid (\mathbf{M}\ \sigma){:}\ \tau_{[\alpha:=\sigma]}}$$

Called the polymorphic λ-calculus or System F. Clumsy in practice. Influenced Java generic 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. ∀ can only appear at the top-level in a type.

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

Typing rules same as extended typed lambda calculus, except for a generalized rule for **`let`**, which we soon discuss.

Extra axiom: Γ, {**`x: ∀T S`**} |– **`x: S'`**
where **`S'`** is any substitution instance of **`S`** (replacing **`x`**). We will also subsequently discuss this rule in more detail.

Implicit Polymorphism cont.

Different instantiations of same type scheme axiom:

$\Gamma, \{\texttt{x: } \forall T\ (T \to T)\} \vdash \texttt{x: int} \to \texttt{int}$

$\Gamma, \{\texttt{x: } \forall T\ (T \to T)\} \vdash \texttt{x: (int} \to \texttt{int)} \to \texttt{(int} \to \texttt{int)}$

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 reclet introduce polymorphic operations!

Defining Polymorphic Functions

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

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

If we type recursive **let** as before, this program is untypable because **id** is used two different ways: as the identity function for type **int**→**int** and for type **int**.

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

$$\frac{\Gamma,\ \mathtt{x}{:}\sigma \mid \mathtt{M}{:}\ \sigma\ ;\quad \Gamma,\ \mathtt{x}{:}\ \mathit{close}(\Gamma,\sigma) \mid \mathtt{N}\ :\ \tau}{\Gamma \mid \mathtt{let\ c := M\ in\ N} : \tau}$$

(non-rec let poly rule)

where *close*(**Γ**,σ) means find all of the free type variables α_1, ..., α_n in σ that do not appear in **Γ** and generate $\forall\alpha_1$, ..., α_n σ.

Note that the typing for **x** in the environment **Γ**, **x**:σ is weaker than it is in the environment **Γ**, **x**: *close*(**Γ**,σ)

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).

This reconstruction process is algorithmic!