Comp 411
Principles of Programming Languages
Lecture 22
Polymorphic Types

Corky Cartwright

March 24, 2021

A Fatal Weakness in Simple Structural Typing

Structural similar types like `list-of-int` and `list-of-bool` are completely disjoint. 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 disjoint types like `int` and `bool`.

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

$$\texttt{int-list ::= empty() | cons}_{\texttt{int}}\texttt{(int, int-list)}$$

$$\texttt{bool-list ::= empty() | cons}_{\texttt{bool}}\texttt{(bool, bool-list)}$$

...

we define a single parameterized form of list:

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

What Types Correspond to Parametric Data?

Henceforth, we will assume that our λ-language supports an arbitrary number of arguments in λ-abstractions, which is required to support pure **let** constructions with an arbitrary number of bindings.

In the data definition:

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

what are the types of data operations like **empty**, **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 $\alpha_1, \ldots, \alpha_n$ ($n > 0$) are type variables, and τ is a type that may include type variables. The types of the data operations in our example are:

```
empty: ∀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 Added to the Type System?

Two Options:

1. First option: *explicit* polymorphism. We add type variables, explicit type abstraction and application to the programming language. The expressions in our language are:

   $$\mathbf{M} ::= \lambda\underline{\mathbf{v}:\boldsymbol{\sigma}}.\ \mathbf{M} \mid (\mathbf{M}\ \underline{\mathbf{N}}) \mid \mathbf{V} \mid \Lambda\underline{\mathbf{t}}.\ \mathbf{M} \mid (\mathbf{M}\ \underline{\boldsymbol{\tau}})$$

   $$\boldsymbol{\tau} ::= \mathbf{D}_1 \mid \ldots \mid \mathbf{D}_n \mid (\underline{\boldsymbol{\sigma}} \rightarrow \boldsymbol{\tau}) \mid \forall\underline{\mathbf{t}}\ \boldsymbol{\tau}$$

   where $\mathbf{V}$ is the set of vars, $\underline{\mathbf{v}:\boldsymbol{\sigma}}$ is a (finite) list of vars together with their types, $\mathbf{T}$ is the set of type variables, $\underline{\mathbf{t}}$ is a finite list of type variables, $\mathbf{M}$ is an expression, $\underline{\mathbf{N}}$ is a (finite) sequence of expressions, $\boldsymbol{\tau}$ is a type expression, and $\underline{\boldsymbol{\sigma}}$ is a finite cross product of type expressions, and $\mathbf{D}_1$ …, $\mathbf{D}_n$ are primitive types. The symbol Λ is a capital λ; it denotes type abstraction, just as λ represents value abstraction. This extension is more general than merely adding type schemes since quantifiers can be embedded within types. It is called the *polymorphic λ-calculus* or *System F*. The typing rules for the simply typed λ-calculus can be extended to type programs in the polymorphic λ-calculus (as shown on the next slide), but type reconstruction (determining the typing, if any, for an untyped expression) is undecidable. The *polymorphic λ-calculus* is clumsy in practice and; it has not been incorporated in any practical programming language. Nevertheless, it is worth understanding because of its impact on the generic type systems for OO languages (like Java 5+).

2. Better option: *implicit* polymorphism (discussed on next few slides)

Typing Rules for the Polymorphic λ-Calculus

- The binding axiom and rules for (functional) abstraction and (functional) application same as in the simply typed λ-calculus (with λ-abstraction generalized to arbitrary finite arity) with one qualification: types may contain type variables.
- Rules for type abstraction and type application:

$$\frac{\Gamma \vdash M{:}\tau;\ \ \underline{\alpha} \text{ not free in } \Gamma}{\Gamma \vdash \Lambda\underline{\alpha}.M{:}\ \forall\underline{\alpha}\ \tau} \qquad \text{(type abstraction rule)}$$

$$\frac{\Gamma \vdash M{:}\ \forall\underline{\alpha}\ \tau}{\Gamma \vdash (M\ \underline{\sigma}){:}\tau_{[\underline{\alpha}:=\underline{\sigma}]}} \qquad \text{(type application rule)}$$

Note: type abstraction and application are degenerate if τ does not contain $\underline{\alpha}$

Implicit Polymorphism

Second option for interpreting type schemes, which is much more important in practice than explicit polymorphism

(*i*) We restrict type expressions to ordinary type expressions (possibly containing type variables) and type schemes which are ordinary type expressions surrounded by a universal quantifier *at the top level*. Hence, $\forall$ can only appear at the top-level in a type expression and the body of a type scheme is an ordinary type.

(*ii*) We make no changes to the programming language, which looks like a (dynamically typed) λ-language.

If we ignore Milner's polymorphic let construct (explained later), the typing rules are the same as for ordinary typed λ-languages, except:

- the inductive definition of types τ includes type variables α as an additional base case;
- the type environment can include type schemes (as defined earlier) in place of types (but type schemes *can only appear in type environments*!); and
- an additional axiom supports fully instantiating type schemes:

  $\Gamma,\ x{:}\forall\underline{\alpha}\ S \vdash x{:}\ S'$

  where S is a type scheme and S' is a substitution instance of S (replacing all type quantified type variables $\underline{\alpha}$) containing no quantifiers.

Note: our typing rules will ensure that in any typing judgment $\Gamma \vdash M{:}\tau$, all variables in M are assigned types in M, all type variables in τ appear free in Γ, and type schemes (ordinary type expressions enclosed by quantification at the top level) only appear in Γ. You should carefully study the Type Inference Study Guide.

Implicit Polymorphism cont.

Different instantiations of same type scheme axiom:

`Γ, x:∀α(α→α) ⊢ x: int→int`

`Γ, x:∀α(α→α) ⊢ x: (int→int) → (int→int)`

In the absence of polymorphic **`let`**, we can use primitive operations with schematic types because the types of primitive operations are built into the base environment (as in Assignment 5), but how do we define new polymorphic operations? We need to extend our language so that **`let`** and **`letrec`** introduce polymorphic operations! Robin Milner introduced these constructions in his original formulation of the language ML, which has spawned a family of statically typed (mostly) functional languages including Haskell and Ocaml. Haskell is purely functional while Ocaml is (mostly) functional.

Typed Jam and Polymorphic Jam

The course master web page contains links to a handout describing two different closely related

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

If we interpret `let` as either pure `let` or recursive `let` as described in our previous lecture, 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 (strengthen) our typing rule for (recursive) `let` as follows:

$$\frac{\Gamma,\ \underline{x:\sigma} \vdash \underline{M:\sigma};\quad \Gamma, \{x:\underline{close(\sigma,\Gamma)}\} \vdash N:\tau}{\Gamma \vdash \mathtt{let}\ \underline{x:\sigma\ := M}\ \mathtt{in}\ N : \tau} \quad \text{(polymorphic let rule)}$$

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

Key intuition: the *proof* of $\Gamma,\ \underline{x:\sigma} \vdash \underline{M:\sigma}$ is *schematic* in the free type variables. Hence, we can instantiate them in the proof without breaking it!

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 interpret **`let`** as either pure **`let`** or recursive **`let`** as described in our previous lecture, this program is untypable because **`id`** is used two different ways: as the identity function for type $\texttt{int} \to \texttt{int}$ and for type **`int`**.

But we can revise (strengthen) our typing rule for (recursive) **`let`** as follows:

$$\frac{\Gamma,\{x{:}\sigma\} \vdash \underline{M{:}\sigma};\quad \Gamma,\{x{:}\underline{close(\sigma,\Gamma)}\} \vdash N{:}\tau}{\Gamma \vdash \texttt{let } x{:}\sigma \texttt{ := } M \texttt{ in } N : \tau} \quad \text{(polymorphic let rule)}$$

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

Key intuition: the proof of $\Gamma,\{x{:}\sigma\} \vdash M{:}\sigma$ is *schematic* in the free type variables in σ. Hence, we can instantiate them in the proof without breaking it!

Type Reconstruction

Implicit polymorphism is far more important in practice than explicit polymorphism because the types in implicitly typed programs 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.) Explicit polymorphism which forces explicit type abstraction and application is painful.

How does type reconstruction work? Mechanically build the type inference tree for a program using the typing rules with type variables for the types of all variables introduced in λ-abstractions. To make this tree a valid proof tree, certain equality relationships must hold between type expressions (these equality constraints are implicit in the rules). Generate the list of equality constraints and solve them (using unification which we will describe in our next lecture).

This reconstruction process is algorithmic! For this reason, *we adopt the convention that types can be dropped from the binding occurrences of variables in implicitly polymorphic λ-languages.* In languages supporting Hindley-Milner typing such annotations are typically optional.