Comp 411
Principles of Programming Languages
Lecture 24
Types for Imperative Languages

Corky Cartwright

March 31, 2023

Does Hindley-Milner Polymorphism Work in Imperative Languages?

The naïve extension of Hindley-Milner Polymorphism to imperative languages fails!

Assume that we add **`ref`** objects and operations to our language. This is purely an extension of the data model, which only involves the definition of types (by adding new type constructors) and the set of primitive operations in our base type environment.

New unary type constructor: **`ref`**

New primitive operations:

```
ref: ∀α (α → ref α)
!: ∀α (ref α → α)
←: ∀α (ref α  α  →  )
```

Breaking the Resulting Type System

Counterexample to sound typing:

```
let x := ref null
in {x <- cons(4,null);
    ~first(!x)}
```

The empty list `null` has type `∀α(list α)`. What is the type of `x`? `ref ∀α(list α)`. Then `x` has type `ref list int` in the first expression of the block and type `ref list bool` in the second. Yet `~first(!x)` will generate a run-time type error because `first(!x)` is an `int`.

What is going wrong? Recall our interpretation of let-polymorphism as a syntactic abbreviation for an appropriate family of non-polymorphic definitions. In this case, the expanded program

```
let x1:(ref list int)  := ref null;
    x2:(ref list bool) := ref null;
    . . .
```

is well-typed! What went wrong in the translation? It changed the meaning of the program because it changed the sharing of `ref` cells.

What Is Fundamentally Different About Imperative Values?

Their semantics involves the concept of *sharing*, which makes reasoning about mathematical expression very messy. Why? Changing the contents of one occurrence of `ref` may change the contents of another because they are shared!

The semantics of function equality in Jam is not purely functional because it relies on testing sharing relationships. A truly functional semantics does not include any notion of sharing between values.

Can We Patch Hindley-Milner Typing So That It Works for Imperative Languages

Yes! It can be done in a variety of ways by imposing additional restrictions on the inference of polymorphic types for program variables (more restrictions on what bindings qualify as polymorphic.

The original “solution” in Standard ML relied on “weak type variables” and was/is generally regarded as incomprehensible. (Look it up on the web if you are interested.) Moreover, many formulations (including the early implementations) of weak type variables are not sound! Soundness proofs for a few variants of this system eventually appeared in the mid-90's (ML dates from 1978) including one by our own John Greiner.

The winning restriction on H-M typing for imperative languages was developed by my late student Andrew Wright (in joint work with Mathias Felleisen).

The Wright-Felleisen Value Test for Polymorphic Generalization

Define a *syntactic* value as either a program variable or a data value (*valu*e in the operational semantics). Rationale: *splitting let bindings must preserve semantics of the let body.* Then the type of a variable introduced in a let construction can be generalized (the close operation in our **let-poly** rule) if and only if the right hand side of the definition is a value.

Why does this work? It is based on the idea that polymorphism only works when the value of a variable can be transparently copied (which is not true in our counterexample) in the replicated bindings. Expressions that mean exactly the same data values can be copied. But *computations* (which generally produce new results) cannot.