Comp 411
Principles of Programming Languages
Lecture 23
Types for Imperative Languages

Corky Cartwright
March 26, 2018
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 (as we did to Jam in Assignment 4). 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.

The new unary type constructor `ref` is supported by the following primitive operations:

\[
\begin{align*}
\text{ref:} & \quad \forall \alpha (\alpha \to \text{ref } \alpha) \\
! & \quad \forall \alpha (\text{ref } \alpha \to \alpha) \\
\leftarrow & \quad \forall \alpha (\text{ref } \alpha \to \alpha)
\end{align*}
\]

But `ref` has a side effect; it allocates a distinct mutable cell (with a unique address).
Breaking the Resulting Type System

Counterexample to sound typing:

```ml
let x := ref null
in {x <- cons(4,null); // x is bound to a int-list
    ~first(!x)}        // x is used as a bool-list
```

The empty list `null` has type $\forall \alpha\text{(list } \alpha)$. What is the type of $x$?

$\text{ref } \forall \alpha\text{(list } \alpha)$ where $\alpha$ occurs free and does not appear in the type environment (which describes types in the enclosing context). Then $x$ has type $\text{ref list int}$ in the first expression of the block and type $\text{ref list bool}$ in the second which are instantiations of the type scheme for $x$. Yet

$\text{~first(!x)}$ generates a run-time type error because $\text{first(!x)}$ is an $\text{int}$.

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

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

This program is well-typed in Typed Jam which is not polymorphic! But what went wrong in the translation? Hindley-Milner type inference is not sound unless we ensure that splitting $\text{let}$ bindings preserves the semantics of programs.
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 among values. Functional Scheme suffers from the same glitch. The ML family of languages avoid this glitch by excluding functions (closures) from the types for which equality is defined. ML only allows equality testing between values of the same *monotype* (ground type). There is no equality operation for function monotypes.

Why do Scheme and Jam support location-based function equality?
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.

The original “solution” in Standard ML relied on “weak type variables” and was/is generally regarded as incomprehensible. 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 John Greiner.

The winning restriction on H-M typing for imperative languages, the value test, was developed by my student Andrew Wright (in joint work with my former colleague Mathias Felleisen).
The Value Test for Polymorphic Generalization

Define a *syntactic value* as either a program variable or a data value (*value* in the operational [reduction] semantics). Then by the *value test* the type of a variable introduced in a *let* construction can be generalized (*close* in *let-poly* rule) if and only if the right hand side of the definition is a *syntactic value*.

Why does this work? It is based on the idea that polymorphism only works when the right-hand-side of a binding can be transparently copied/split (which is not true in our counterexample). Recall our reduction of parametric polymorphism to non-polymorphic form by splitting each polymorphic definition within a *let* into $k$ monomorphic definitions each of which is used once in the body of the *let*. Data values and simple references to data values can be copied. But computations (which generally produce new results) cannot.