CS655 Lecture 17 | Second-Order Type Systems | 28 March, 2006

Scribe: Nora Sovarel | Lecturer: Wes Weimer

Today's class focused on introducing a second-order type system, F_2. Wes started by presenting the limitations of first-order type systems. Next, he presented the features needed for a second-order type system. Two attempts to build such a language prooved not restrictive enough and, finally, the third attempt was just about right, allowing exactly what we need but not more.

1 Limitations of F1

1.1 Review

F_1 supports: function types (simply typed λ-calculus), simple types (integers and booleans), structured types (products and sums), imperative types (references and exceptions), recursive types.

1.2 Limitations

The fundamental limitation of F_1 is that a function works for exactly one type, in other words, one function has exactly one type. A trivial example is the identity function. In F_1 we need a distict identity function for each type, even though its behavior and implementation is the same for all of them: return the given parameter. A more motivating example is sort function. We would like to implement it once and use it for different element types of the collection to be sorted. The sort functions would perform exactly the same operations at runtime, but different versions are needed to keep the type checker happy. Typically, a generic sort function is given the comparison function as a parameter. The collection elements have a generic type, like void* in C (circumvent the type system) or an Object instance in Java (more flexible type system). We want to have only one sort function, more precisely, we want no specialization depending on type.

2 Polymorphism

The solution to the limitations of F_1 is to extend the type system. In this section we will introduce the notion of polymorphism.

An informal definition of polymorphism: "A function is polymorphic if it can be applied to *many* types of arguments". We distinguish between the following types of polymorphism:

- subtype polymorphism (aka bounded polymorphism) where "many types" is all subtypes of a given type. This type of polymorphism is present in object oriented languages like C++ and Java. Its advantage is that is easy to reason about it.
- ad-hoc polymorphism where "many types" depends on the function. The behavior is chosen at runtime (depending on types, e.g. sizeof). Here, the programmer plays the role of the type system verifier.
- parametric predicative polymorphism where "many types" means all monomorphic types. This type of polymorphism is used in practice. The parameter is a type parameter or a type variable. For example, a length function can be defined as $int\ list \rightarrow \alpha\ list$. Here α is a type variable. Its value can be only a monomorphic type.
- parametric impredicative polymorphism where "many types" means all types.

The next sections will describe the two types of parametric polymorphism.

3 Impredicative Parametric Polymorphism

Type variable: $\tau ::= b \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\tau$. $\forall t.\tau$ means t ranges over all types.
Type expression: $e ::= x \mid \lambda x{:}\,\tau.\, e \mid e_1 e_2 \mid \Lambda t.\, e \mid e[\tau]$.

$\Lambda t.\, e$ is type abstraction or type generalization. It means for all types.

$e[\tau]$ is type application or instantiation.

We will walk through the example of identity function.

Type expression: $\Lambda t.\, \lambda x{:}\, t.\, x$

Type: $\forall t.t$

Identity function type applied to the int type: $id[int]$.

The type of identity function applied to int: $int \to int$.

Impredicative typing rules:

$$\frac{x : \tau \ in \ \Gamma}{\Gamma \vdash x{:}\,\tau}$$

$$\frac{\Gamma, x{:}\,\tau \vdash e{:}\,\tau'}{\Gamma \vdash \lambda x{:}\,\tau.e{:}\,\tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1{:}\,\tau \quad \Gamma \vdash e_2{::}\,\tau}{\Gamma \vdash e_1 e_2{:}\,\tau'}$$

$$\frac{\Gamma \vdash e{:}\,\tau}{\Gamma \vdash \Lambda t.e{:}\,\forall t.\tau} \ t \ does \ not \ occur \ in \ \Gamma \ (new \ type \ variable)$$

Note: The side rule prevents ill-formed terms like $\lambda x{:}\,t.\Lambda t.x$

$$\frac{\Gamma \vdash e{:}\,\forall t.\tau'}{\Gamma \vdash e[\tau]{:}\,[\tau/t]\tau'} \ (\ new \ rule)$$

The evaluation rules are the same as those of F_1.

The impredicative polymorphism is useful because one can prove properties of a term just from its type. Examples of such theorems are: there is only one value of type $\forall t.t \to t$, there is no value of type $\forall t.t$. A more motivating example is reverse function, $\forall t.t\ List \to t\ List$. Inside the reverse function the type is not known so no new elements can be added to the list and elements cannot be copied. Hence, the reverse function can produce at most a subset or a permutation of the original list.

The impredicative polymorhpism is extremely expressive. We can encode base types, booleans and naturals. Furthermore, for boolean type we can prove there are only two values. In addition, we can encode disjoint types, product types, unit type. Primitive recursion can be encoded, but not full recursion. We cannot write interesting programs since all terms in F_2 have a termination proof in second-order Peano arithmetic.

The syntax of F_2 is very simple, but the semantics is very complicated. Let's consider the example of identity function applied to itself, $id[\forall t.t \to t]\ id$. The set-theoretic interpretation of id would be a function whose domain contains a set that contains id; thus the semantics of F_2 cannot always be explained intuitively. Type reconstruction (i.e., type inference) is undecidable if type application and abstraction are missing. This means that one cannot tell if an unannotated program typechecks.

4 Predicative Parametric Polymorphism

Our first attempt to fix the problems presented at the end of the previous section is to introduce a restriction: type variables can be instantiated only with monomorphic types. Type variables cannot range over $\forall$-types.

The syntax of predicative polymorphism:

Monomorphic types: $\tau ::= b \mid \tau_1 \to \tau_2 \mid t$

Polymorphic types: $\sigma ::= \tau \mid \forall t.\sigma \mid \sigma_1 \to \sigma_2$

Type expression: $e::=x \mid e_1 e_2 \mid \lambda x{:}\sigma.e \mid \Lambda t.e \mid e[\tau]$

Note the type application expression: a type expression can be applied only to monomorphic types. Now we cannot apply id to itself anymore, it is syntactically impossible. However, we can apply the identity function to a particular identity function, like the identity function for integers.

The typing rules are the same with the ones of impredicative polymorphism. The semantics and termination proofs are simpler. Type reconstruction is still undecidable.

4.1 Prenex Predicative Polymorphism

To make type reconstruction decidable a new restriction is introduced: the polymorphic type constructor $\forall$ appears only at the top level of a type.

Monomorphic types: $\tau::=b \mid \tau_1 \to \tau_2 \mid t$

Polymorphic types: $\sigma::=\tau \mid \forall t.\sigma$

Type expression: $e::=x \mid e_1 e_2 \mid \lambda x{:}\sigma.e \mid \Lambda t.e \mid e[\tau]$

Note that $\forall$ can appear only at the begining of the type. Now $(\forall t.\ t \to t) \to (\forall t.\ t \to t)$ is not a valid type. The type $\forall t.\forall t'.(t \to t) \to (t' \to t')$ is valid, but is not the same as $(\forall t.\ t \to t) \to (\forall t.\ t \to t)$.

This restricted form of predicative polymorphism is called prenex predicative polymorphism. The typing rules are the same. The semantics and termination proofs are still simple. The problem is that we have simplified too much. Prenex predicative polymorphism is not expressive enough to encode booleans and naturals. This is not a major practical concern, but the fact that a formal argument cannot be polymorphic is. Consider the example of a function with a sorting function as a formal argument. The sorting function cannot be used with different types.

The ML solution is a slight extention of F_2: let with a polymorphic type. Type reconstruction remains decidable and it also allows for polymorphic uses of sort.

Let syntax: $let\ x{:}\sigma\ =\ e_1\ in\ e_2$

Let semantics: $(\lambda x{:}\sigma.e_2)e_1$

Type: $[e_1/x]e_2$

$$\frac{\Gamma \vdash e_1{:}\sigma \quad \Gamma, x{:}\sigma \vdash e_2{:}\tau}{\Gamma \vdash let\ x{:}\sigma = e_1\ in\ e_2{:}\tau}$$

This extension was a major design flaw in ML. The let is evaluated using call-by-value, but is typed using call-by name. Unsafe example:

$let\ x{:}\forall t.(t \to t) ref = \Lambda t.ref(\lambda t.x)$
in
  $x[bool]{:}= \lambda x{:}bool.not\ x;$
  $(!x[int])5$

The function stored has type $bool \to bool$, but at runtime it is treated as an $int \to int$ and it is applied to int parameters.

To fix this problem the let extention is restricted to syntactic values only. The difference is that evaluation of e_1 cannot have side effects. Given that restriction, call-by-value and call-by-name behive identically.

5 Subtype Bounded Polymorphism

This type of polymorphism is widely used in practice. The instances of a given type variable are bounded: $\forall t \prec \tau.\sigma$

Discussion:

Let's consider two functions: $f{:}\forall t \prec \tau.t \to \sigma$ and $f'{:}\tau \to \sigma$. Function f can be invoked on type τ and all its subtypes, but f' can be invoked only on type τ.

Let's consider two functions: $f{:}\forall t \prec \tau.t \to t$ and $f'{:}\tau \to \tau$. If $x : \tau' \prec \tau$ then $f[\tau]x : \tau'$ and $f'x : \tau$. We lost information with f' and we may need runtime checks to recover it safely.