CMSC 336: Type Systems for Programming Languages

Lecture 11: Existential Types

Acar & Ahmed **21 February 2008**

Contents

1 Impredicative, Predicative, and Prenex Polymorphism

Impredicative Polymorphism. Consider the encoding of booleans in System F.

$$\begin{aligned} \texttt{bool} &\equiv \forall\alpha.\alpha \to \alpha \to \alpha \\ \texttt{true} &\equiv \Lambda\alpha.\forall x:\alpha.\forall y:\alpha.x \\ \texttt{false} &\equiv \Lambda\alpha.\forall x:\alpha.\forall y:\alpha.y \\ \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 &\equiv e_0[\tau]e_1e_2 \end{aligned}$$

Note that in the encoding of `if` we instantiate a universal type (the type of e_0) with the result type. The result type itself could be a `bool`. This makes the types larger. This property of System F is call *impredicativity*, and we say that System F supports *impredicative polymorphism*, or alternatively, System F allows *impredicative (type) quantification*.

The type system is powerful enough that you can have types that get bigger. This is essentially where the power of these type system comes from.

Another example. Consider

$$\tau := \forall\alpha.\alpha \to \alpha.$$

Consider now an expression $e : \tau$ applied to τ itself.

$$\frac{\vdash e : \forall\alpha.\alpha \to \alpha \qquad \vdash \tau}{\vdash e\ [\tau] : (\alpha \to \alpha)[\tau/\alpha]}$$

Hence, the type of $e\ [\tau]$ is bigger than the type of e:

$$\begin{aligned} & e\ [\tau] \ : \ (\alpha \to \alpha)[\tau/\alpha] \\ \equiv\ & e\ [\tau] \ : \ (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha) \end{aligned}$$

Compare this to what happens with application in simply typed lambda calculus.

$$e : \tau_1 \ \rightarrow \ \tau_2$$

Now, the application $e\ v$ has type τ_2 (assuming that v has type τ_1). So in the simply-typed lambda calculus, application does not make the types bigger.

In System F, this ability for types to get larger and larger allows us to encode things like numbers which can be "unbounded" using universal types.

Predicative Polymorphism. The alternative to impredicative quantification, called *predicative (type) quantification*, is to allow an expression e of type $\forall\alpha.\tau$ to be applied only *un-quantified* types, or types that are quantifier-free. For instance, we may apply $e : \forall\alpha.\alpha \ \rightarrow \ \alpha$ to the type $\texttt{int} \ \rightarrow \ \texttt{int}$ (where `int` is a primitive type), which means that the expression $e\ [\texttt{int} \ \rightarrow \ \texttt{int}]$ would have type $(\texttt{int} \ \rightarrow \ \texttt{int}) \ \rightarrow \ (\texttt{int} \ \rightarrow \ \texttt{int})$, which is "bigger" than the type $\forall\alpha.\alpha \ \rightarrow \ \alpha$ in one sense (since it has more symbols), but it's "smaller" in another sense (it has fewer quantifiers). The predicative fragment of the language is less expressive than the impredicative System F.

Prenex Polymorphism: For the sake of type inference, languages like ML only permit an even more restricted form of polymorphism, called *prenex polymorphism* that allows quantifiers to occur only at the outermost level of a type. The prenex fragment can be formalized as follows, by stratifying types into two parts: *monotypes* (which do not involve and quantification) and *polytypes* (which include the monotypes and permit quantification over monotypes).

$$\begin{array}{llll} \text{Mono} & \tau & ::= & \alpha \mid \tau_1 \ \rightarrow \ \tau_2 \\ \text{Poly} & \sigma & ::= & \tau \mid \forall\alpha.\sigma \end{array}$$

As mentioned above, and obvious from the above grammar of types, in prenex polymorphism all quantifiers must occur at the beginning (outermost level) of a type.

2 Existential Types

Consider an expression e of universal type $\forall\alpha.\tau$.

The logical interpretation of this is that if you have any type σ (i.e., for all types σ), then the expression e must have that type, i.e., $\tau[\sigma/\alpha]$.

Operationally, we think of an expression of universal type as a suspended computation $\Lambda\alpha.e$ that when applied to any type σ would give us an expression that is customized for that type.

We have discussed one form of quantified types. In the rest of this lecture, we'll discuss *existential types* $\exists\alpha.\tau$.

Consider an expression e that has an existential type $\exists\alpha.\tau$.

The logical interpretation of this is that there exists *some* type σ such that $e : \tau[\sigma/\alpha]$.

We will be more interested in the operational interpretation. Operationally, you can think of e this as a pair that consists of a *witness type* σ (that is hidden by α in the existential type $\exists\alpha.\tau$), and an expressions e_1 of type $\tau[\sigma/\alpha]$.

Existential types essentially capture the notion of abstraction—they hide the witness type.

We now extend System F with existential types:

$$\tau \quad ::= \quad \ldots \mid \exists\alpha.\tau$$

The introduction form for existential types is $\texttt{pack}\ \tau\ \texttt{with}\ e$. Some examples with `pack`:

$$\begin{array}{lcl} \texttt{pack nat with } \{a = 5, f : \lambda x : \texttt{nat}.succ(x)\} & : & \exists\alpha.\{a : \texttt{nat}, f : \texttt{nat} \ \rightarrow\ \texttt{nat}\} \\ & : & \exists\alpha\{a : \alpha, f : \alpha \ \rightarrow\ \alpha\} \\ & : & \exists\alpha\{a : \alpha, f : \alpha \ \rightarrow\ \texttt{nat}\} \end{array}$$

Notice that we can give many different types to a `pack` expression. Since the existential type that can be ascribed to a `pack` expression is ambiguous, we have to specify what type to ascribe to a `pack` expression. Thus, the introduction form for existential types will have the following form: $\texttt{pack}\ \sigma\ \texttt{with}\ e\ \texttt{as}\ \exists\alpha.\tau$.

Typechecking `pack`:

$$\frac{\Delta \vdash \sigma\ \texttt{type} \quad \Delta;\Gamma \vdash e : \tau[\sigma/\alpha]}{\Delta;\Gamma \vdash \texttt{pack}\ \sigma\ \texttt{with}\ e\ \texttt{as}\ \exists\alpha.\tau : \exists\alpha.\tau}$$

Examples:

$\texttt{pack nat with } 0 \texttt{ as } \exists\alpha.\alpha : \exists\alpha.\alpha$
$\texttt{pack bool with true as } \exists\alpha.\alpha : \exists\alpha.\alpha$
$\texttt{pack bool with } \{a = \texttt{true}, f = \lambda x : \texttt{bool}.1\} \texttt{ as } \exists\alpha.\{a : \alpha, f : \alpha \ \rightarrow\ \texttt{nat}\}$

The elimination form for existential types is $\texttt{unpack}\ e_1\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2$.

$$e \quad ::= \quad \ldots \mid \texttt{pack}\ \sigma\ \texttt{with}\ e\ \texttt{as}\ \exists\alpha.\tau \mid \texttt{unpack}\ e_1\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2$$

The typing rule for `unpack`:

$$\frac{\Delta;\Gamma \vdash e_1 : \exists\alpha.\tau \quad \Delta \vdash \tau_2\ \texttt{type} \quad \Delta, \alpha\ \texttt{type};\Gamma, x : \tau \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \texttt{unpack}\ e_1\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2 : \tau_2}$$

Examples: Consider the following:

$$p1 = \texttt{pack nat with } \{a = 5, f : \lambda x : \texttt{nat}.succ(x)\} \texttt{ as } \exists\alpha\{a : \alpha, f : \alpha \ \rightarrow\ \texttt{nat}\}$$

Note that $p1 : \exists\alpha\{a : \alpha, f : \alpha \ \rightarrow\ \texttt{nat}\}$.

- The following expression type checks:

$$\texttt{unpack}\ p1\ \texttt{as}\ \alpha, x\ \texttt{in}\ x.f\ x.a \ : \ \texttt{nat}$$

- The following does not type check since $x.f$ expects an argument of type α instead of `nat`.

$$\texttt{unpack}\ p1\ \texttt{as}\ \alpha, x\ \texttt{in}\ x.f\ \texttt{zero}\ :\ \texttt{Error}$$

- The following type checks—notice that α appears in the body of the `unpack`:

$$\texttt{unpack}\ p1\ \texttt{as}\ \alpha, x\ \texttt{in}\ (\lambda y : \alpha.\ x.f\ y)\ x.a\ :\ \texttt{nat}$$

- The following does not type check since $x.a$ is of type α, whereas `succ` $-$ expects a `nat`:

$$\texttt{unpack}\ p1\ \texttt{as}\ \alpha, x\ \texttt{in}\ \texttt{succ}\ x.a\ :\ \texttt{Error}$$

- The following does not type check since the body of the unpack has type α—but the typing rule for unpack requires that α not occur free in the result type of the body:

$$\texttt{unpack}\ q1\ \texttt{as}\ \alpha, x\ \texttt{in}\ x.a\ :\ \texttt{Error}$$

3 Dynamic Semantics

What are the values of existential type? Any package that packs a concrete type with a value is a value.

$$v \quad ::= \quad \ldots \mid \texttt{pack}\ \sigma\ \texttt{with}\ v\ \texttt{as}\ \exists\alpha.\tau$$

$$\frac{e\ \rightarrow\ e'}{\texttt{pack}\ \sigma\ \texttt{with}\ e\ \texttt{as}\ \exists\alpha.\tau\ \rightarrow\ \texttt{pack}\ \sigma\ \texttt{with}\ e'\ \texttt{as}\ \exists\alpha.\tau}$$

$$\frac{e_1\ \rightarrow\ e_1'}{\texttt{unpack}\ e_1\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2\ \rightarrow\ \texttt{unpack}\ e_1'\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2}$$

$$\frac{}{\texttt{unpack}\ (\texttt{pack}\ \sigma\ \texttt{with}\ v\ \texttt{as}\ \exists\alpha.\tau)\ \texttt{as}\ \alpha, x\ \texttt{in}\ e_2\ \rightarrow\ e_2[\sigma/\alpha][v/x]}$$

4 CLU Example

```
ADT Counter =
  type counter              // abstract type name
  representation nat        // concrete type
  signature
    c: counter
    get: counter -> nat
    inc: counter -> counter

  operations
    c = 0
    get = \ x: nat. x
    inc = \ x: nat. x+1
```

CLU is from the 70's. In a paper entitled "Abstract Types have Existential Type," Mitchell and Plotkin [POPL 1988] showed that abstract types declarations in languages like CLU, Ada, and ML could be encoded using existential types.

The signature part of the above ADT can be encoded as:

$$\texttt{COUNTER} = \exists \texttt{counter}.\, \{c : \texttt{counter}, \mathit{get} : \texttt{counter} \to \texttt{nat}, \mathit{inc} : \texttt{counter} \to \texttt{counter}\}$$

The operations part of the above ADT can be encoded as follows:

$$\mathit{cntr1} = \texttt{pack nat with } \{c = 0, \mathit{get} = \lambda x : \texttt{nat}.x, \mathit{inc} = \lambda x : \texttt{nat}.x + 1\} \texttt{ as COUNTER}$$

Examples of use of *cntr1*:

$$\texttt{unpack } \mathit{cntr1} \texttt{ as } \alpha, x \texttt{ in } x.\mathit{get}\ (x.\mathit{inc}\ x.c)$$

$$\begin{array}{l}\texttt{unpack } \mathit{cntr1} \texttt{ as } \alpha, x \texttt{ in} \\ \quad \texttt{let } \mathit{add3} = \lambda c : \alpha.\ x.\mathit{inc}\ (x.\mathit{inc}\ (x.\mathit{inc}\ c)) \texttt{ in} \\ \quad\quad x.\mathit{get}\ (\mathit{add3}\ x.c)\end{array}$$

5 Representation Independence

Representation independence is the property that the behavior of a client cannot depend on the particular concrete representation of an existential type. For instance, consider the following implementation which also has the type COUNTER:

$$\mathit{cntr2} = \texttt{pack nat with } \{c = 0, \mathit{get} = \lambda x : \texttt{nat}. - x, \mathit{inc} = \lambda x : \texttt{nat}.x - 1\} \texttt{ as COUNTER}$$

Notice that *cntr1* and *cntr2* are *observationally equivalent*—that is, no client will be able to tell the difference between them.