CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Existential Types

Existentials and Type Abstraction

What might a type like $\exists\alpha.\tau$ mean?

If we think of types as representing **sets** (or domains) of values, then we might expect that $v \in \exists\alpha.\tau$ implies that there exists **some** τ' such that $v \in \tau[\tau'/\alpha]$.

But the pragmatic use of existentials is based on a more operational intuition: if $v : \exists\alpha.\tau$, then v represents a **pair** consisting of a **type** τ' and a **value** v' such that $v' : \tau[\tau'/\alpha]$.

An existential package is essentially like a **module** with a **hidden** type component and an expression component (normally a record of operators), wrapped up in an interface type. Clients of the package can use only the operators in the interface; they cannot "see" the hidden type.

It is important to realize that existentials types are just formal type constructions like any other. They arise (only) by use of an explicit **introduction** rule, and can be broken down using an explicit **elimination** rule. Each rule is associated with a coercion operator in the underlying expression language.

Introduction Form: Pack

$$\texttt{package = pack [} \textit{hidden-type}\texttt{,}\ \textit{implementation}\texttt{]}\ \texttt{as}\ \exists\alpha\texttt{.}\ \textit{signature}$$

Examples:

$$\texttt{p1 = pack [Int, 0] as } \exists\alpha.\alpha$$

$$\texttt{p2 = pack [Int, \{ a = 0, f = }\lambda\texttt{x:Int.x+a \}]}\ \texttt{as } \exists\alpha.\{a:\alpha, f:\alpha\rightarrow\alpha\}$$

$$\texttt{p3 = pack [Int, \{ a = 0, f = }\lambda\texttt{x:Int.x+a \}]}\ \texttt{as } \exists\alpha.\{a:\alpha, f:\alpha\rightarrow \texttt{Int}\}$$

$$\texttt{p4 = pack [Bool, \{ a = True, f = }\lambda\texttt{x:Bool. a||x ? 1 : 0 \}]}\ \texttt{as } \exists\alpha.\{a:\alpha, f:\alpha\rightarrow \texttt{Int}\}$$

$$\texttt{p5 = pack [Int, \{ a = 0, f = }\lambda\texttt{x:Int.x+a \}]}\ \texttt{as } \exists\alpha.\{a:\texttt{Int}, f:\texttt{Int}\rightarrow \texttt{Int}\}$$

(1) is useless; (2) and (3) have same implementation but different signatures; (3) and (4) have same signature but different implementations; (5) doesn't hide anything.

$$\frac{E \ \triangleright M : \tau[\tau'/\alpha]}{E \ \triangleright \texttt{pack [}\tau'\texttt{,}M\texttt{] as } \exists\alpha.\tau : \exists\alpha.\tau}\ PACK$$

Elimination Form: Open

open *package* as [*hidden-type-name*,
implementation-name]
in *client-body*

Uses of an existential value must be restricted so that they can be type-checked without depending on the actual implementation type – which might not always be the same, or be known until runtime.

More subtly, hidden type name must not escape the client body, lest it be unified with some known type.

Examples, supposing `p :` $\exists\alpha.\{a : \alpha, f : \alpha \to \texttt{Int}\}$:

```
open p as [β,r] in
   r.f (r.a) + 2
```

```
open p as [β,r] in
  (λy:β. r.f y) (r.a) + 2
```

```
open p as [β,r] in
   r.a + 2
```
Fails to type!!

```
(open p as [β,r] in
   r.a) + 2
```
Fails to type!!

$$\frac{E \rhd M : \exists\alpha.\tau' \quad E + \{x : \tau'[\beta/\alpha]\} \rhd N : \tau \quad \beta \notin FV(\tau)}{E \rhd \texttt{open}\ M\ \texttt{as}\ [\beta,x]\ \texttt{in}\ N : \tau}\ OPEN$$

Semantics

If

`p = pack [`τ'`,`M`] as` $\exists\alpha.\tau$

then

`open p as [`β`,`x`] in` N

reduces to

$N[M/x][\tau'/\beta]$

and is operationally just like

`let` x `=` M `in` N

Abstract Data Types

Existential package corresponds to an abstract data type (ADT) implementation.

We can **parameterize** client code by the implementation; existential typing ensures that it works equally well regardless of the underlying hidden type and implementation.

Example:

```
p3 = pack [Int, { a = 0, f = λx:Int.x+a }]
               as ∃α.{a : α, f : α → Int}

p4 = pack [Bool, { a = True,
                   f = λx:Bool. a||x ? 1 : 0 }]
               as ∃α.{a : α, f : α → Int}

client = λ p:∃α.{a : α, f : α → Int}.
              open p as [β,pp] in
                  pp.f (pp.a) + pp.f(pp.a)


client3 = client p3

client4 = client p4
```