Existential Types

Klaus Ostermann
Aarhus University

Existential Types

- Are „dual“ to universal types
- Foundation for data abstraction and information hiding
- Two ways to look at an existential type {∃X,T}
  - Logical intuition: a value of type T[X:=S] for some type S
  - Operational intuition: a pair {*S,t} of a type S and term t of type T[X:=S]
- Other books use the (more standard) notation ∃X.T. We stick to Pierce‘s notation {∃X,T}

Building and using terms with existential types

- Or, in the terminology of natural deduction, *introduction* and *elimination* rules
- Idea: A term can be packed to hide a type component, and unpacked (or: openend) to use it

Example

```
counterADT =
   {*Nat,
    {new = 1,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};

▸ counterADT : {∃Counter,
                  {new:Counter,get:Counter→Nat,inc:Counter→Counter}}

 let {Counter,counter} = counterADT in
 counter.get (counter.inc counter.new);

▸ 2 : Nat
```

Example

```
let {Counter,counter}=counterADT in
let add3 = λc:Counter. counter.inc (counter.inc (counter.inc c)) in
counter.get (add3 counter.new);
```

```
▸ 4 : Nat
```

```
let {Counter,counter} = counterADT in

let {FlipFlop,flipflop} =
     {*Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop, read: FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop, reset: FlipFlop→FlipFlop}} in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
```

```
▸ false : Bool
```

Existential Types

New syntactic forms

t	::=	...	*terms:*
		`{*T,t} as T`	*packing*
		`let {X,x}=t in t`	*unpacking*
v	::=	...	*values:*
		`{*T,v} as T`	*package value*
T	::=	...	*types:*
		`{∃X,T}`	*existential type*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\text{let }\{X,x\}=(\{{*}T_{11},v_{12}\}\text{ as }T_1)\text{ in }t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2 \quad \text{(E-UnpackPack)}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{{*}T_{11},t_{12}\}\text{ as }T_1 \longrightarrow \{{*}T_{11},t'_{12}\}\text{ as }T_1} \quad \text{(E-Pack)}$$

$$\frac{t_1 \longrightarrow t'_1}{\text{let }\{X,x\}=t_1\text{ in }t_2 \longrightarrow \text{let }\{X,x\}=t'_1\text{ in }t_2} \quad \text{(E-Unpack)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{{*}U,t_2\}\text{ as }\{\exists X,T_2\} : \{\exists X,T_2\}} \quad \text{(T-Pack)}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X,T_{12}\} \qquad \Gamma, X, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let }\{X,x\}=t_1\text{ in }t_2 : T_2} \quad \text{(T-Unpack)}$$

Encoding existential types by universal types

- In logic we have $\neg\, \exists x{\in}\mathbf{X}\, P(x) \equiv \forall x{\in}\mathbf{X}\, \neg P(x)$
- We can simulate an existential type by a universal type and a "continuation"

$$\{\exists X,T\} \stackrel{\text{def}}{=} \forall Y.\ (\forall X.\ T \to Y) \to Y.$$

- Recall that, via Curry-Howard, CPS transformation corresponds to double negation!

Encoding existential types by universal types

- Packing

$$\{*S,t\} \text{ as } \{\exists X,T\} \stackrel{\text{def}}{=} \lambda Y.\ \lambda f{:}(\forall X.T\to Y).\ f\ [S]\ t$$

- Unpacking

$$\text{let } \{X,x\}=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} t_1\ [T_2]\ (\lambda X.\ \lambda x{:}T_{11}.\ t_2).$$

Forms of existential types: SML

```
signature INT_QUEUE = sig
  type t
  val empty : t
  val insert : int * t -> t
  val remove : t -> int * t
end
```

Forms of existential types: SML

```
structure IQ :> INT_QUEUE = struct
  type t = int list
  val empty = nil
  val insert = op ::
  fun remove q =
    let val x::qr = rev q
     in (x, rev qr) end
end
structure Client = struct
  … IQ.insert … IQ.remove …
end
```

Open vs closed Scope

- Existentials via pack/unpack provide no direct access to hidden type **(closed scope)**
  - If we open an existential package twice, we get two different abstract types!
- If S is an SML module with hidden type t, then each occurrence of S.t refers to the same unknown type
  - SML modules are not first-class whereas pack/unpack terms are

Forms of existential types: Java Wildcards

```
Box<?>                  ⟶  ∃X.Box<X>
Box<Box<?>>             ⟶  Box<∃X.Box<X>>
Box<? extends Dog>      ⟶  ∃X◁Dog.Box<X>
Pair<?,?>               ⟶  ∃X.∃Y.Pair<X,Y>
```

From: “Towards an Existential Types Model for Java Wildcards”, FTFJP 2007

```
void m1(Box<?> x) {...}
void m2(Box<Dog> y) { this.m1(y); }
```

is translated to:

```
void m1(∃X.Box<X> x) {...}
void m2(Box<Dog> y)} { this.m1(close y with X hiding Dog); }
```

```
<X>Box<X> m1(Box<X> x) {...}
Box<?> m2(Box<?> y) { this.m1(y); }
```

is translated to (note how opening the existential type allows us to provide an actual type parameter to m1):

```
<X>Box<X> m1(Box<X> x) {...}
∃Z.Box<Z> m2(∃Y.Box<Y> y) {
  open y,Y as y2 in
     close
       this.<Y>m1(y2)     \\has type Box<Y>
     with Z hiding Y;     \\has type ∃Z.Box<Z>
}
```

Forms of existential types:

Existentially quantified data constructors in Haskell

```
data Obj = forall a. (Show a) => Obj a

xs :: [Obj]
xs = [Obj 1, Obj "foo", Obj 'c']

doShow :: [Obj] -> String
doShow [] = ""
doShow ((Obj x):xs) = show x ++ doShow xs
```