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 \( \{\exists X, T\} \)
  – 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 \( \exists X.T \). We stick to Pierce‘s notation \( \{\exists X, T\} \)
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
counterADT =
  {*Nat,
   {new = 1,
    get = \i:Nat. \i,
    inc = \i:Nat. succ(i)}}
as {\exists Counter,
  {new: Counter,
    get: Counter->Nat,
    inc: Counter->Counter}};

▷ counterADT : {\exists 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 = \c: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 = \c:Counter. iseven (counter.get c),
    toggle = \c:Counter. counter.inc c,
    reset = \c:Counter. counter.new} as {\exists 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

\[ t ::= \ldots \]
\[ \{\ast T, t\} \text{ as } T \]
\[ \text{let } \{X,x\}=t \text{ in } t \]

\[ v ::= \ldots \]
\[ \{\ast T, v\} \text{ as } T \]

\[ T ::= \ldots \]
\[ \{\exists X, T\} \]

terms:

packing

unpacking

values:

package value

types:

existential type

New evaluation rules

\[ \text{let } \{X,x\}=(\{\ast T_{11}, v_{12}\} \text{ as } T_1) \text{ in } t_2 \]
\[ \rightarrow [X \rightarrow T_{11}][x \rightarrow v_{12}]t_2 \] (E-UNPACKPACK)

\[ t_{12} \rightarrow t'_{12} \]
\[ \{\ast T_{11}, t_{12}\} \text{ as } T_1 \]
\[ \rightarrow \{\ast T_{11}, t'_{12}\} \text{ as } T_1 \] (E-PACK)

\[ t_1 \rightarrow t'_1 \]
\[ \text{let } \{X,x\}=t_1 \text{ in } t_2 \]
\[ \rightarrow \text{let } \{X,x\}=t'_1 \text{ in } t_2 \] (E-UNPACK)

New typing rules

\[ \Gamma \vdash t : T \]

(T-PACK)

\[ \Gamma \vdash t_2 : [X \rightarrow U]T_2 \]
\[ \Gamma \vdash \{\ast U, t_2\} \text{ as } \{\exists X, T_2\} : \{\exists X, T_2\} \] (T-PACK)

\[ \Gamma \vdash \text{let } \{X,x\}=t_1 \text{ in } t_2 : T_2 \] (T-UNPACK)
Encoding existential types by universal types

• In logic we have $\neg \exists x \in X \ P(x) \equiv \ \forall x \in X \ \neg P(x)$

• We can simulate a existential types by a universal type and a “continuation”

  \[ \{ \exists X, T \} \overset{\text{def}}{=} \ \forall Y. \ (\forall X. \ T \rightarrow Y) \rightarrow Y. \]

• Recall that, via Curry-Howard, CPS transformation corresponds to double negation!
Encoding existential types by universal types

- Packing

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

- Unpacking

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

From: “Towards an Existential Types Model for Java Wildcards”, FTFJP 2007
Forms of existential types:
Existentially quantified data constructors in Haskell

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

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

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