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
Example

counterADT =
{ *Nat,
  {new = 1,
    get = \lambda i: \text{Nat}. \ i,
    inc = \lambda i: \text{Nat}. \ \text{succ}(i)}
  }

as {\exists \text{Counter},
  {new: \text{Counter},
    get: \text{Counter} \rightarrow \text{Nat},
    inc: \text{Counter} \rightarrow \text{Counter}}};

counterADT : {\exists \text{Counter},
  {new: \text{Counter}, get: \text{Counter} \rightarrow \text{Nat}, inc: \text{Counter} \rightarrow \text{Counter}}}

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

\triangleright 2 : \text{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

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

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

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

terms:

packing

unpacking

values:

package value

types:

existential type

New evaluation rules

\[ t \rightarrow t' \]
\[ \text{let } \{X, x\} = (\{\star T_{11}, v_{12}\} \text{ as } T_{11}) \text{ in } t_2 \]
\[ \rightarrow [X \rightarrow T_{11}][x \rightarrow v_{12}] t_2 \]

(E-UNPACKPACK)

New typing rules

\[ \Gamma \vdash t : T \]

(T-PACK)

\[ \Gamma \vdash \{\star U, t_2\} \text{ as } \{\exists X, T_2\} \]
\[ : \{\exists X, T_2\} \]

(E-UNPACK)

\[ \Gamma \vdash t_1 : \{\exists X, T_{12}\} \]
\[ \Gamma, X : T_{12} \vdash t_2 : T_2 \]
\[ \Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2 \]

(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 an existential type 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

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

• Unpacking

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

\[
\begin{align*}
\text{Box}\langle ? \rangle & \quad \longrightarrow \quad \exists X.\text{Box}\langle X \rangle \\
\text{Box}\langle \text{Box}\langle ? \rangle \rangle & \quad \longrightarrow \quad \text{Box}\langle \exists X.\text{Box}\langle X \rangle \rangle \\
\text{Box}\langle ? \rangle & \text{ extends Dog} & \quad \longrightarrow \quad \exists X.\langle \text{Dog.}\text{Box}\langle X \rangle \rangle \\
\text{Pair}\langle ?,? \rangle & \quad \longrightarrow \quad \exists X.\exists Y.\text{Pair}\langle X,Y \rangle \\
\end{align*}
\]

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

is translated to:
void m1(\exists X.\text{Box}\langle X \rangle \ x) {...}
void m2(\text{Box}\langle \text{Dog} \rangle \ y) \ { \ this.m1(\text{close } y \ \text{with } X \ \text{hiding } \text{Dog}); \ }

\langle X \rangle \text{Box}\langle X \rangle \ m1(\text{Box}\langle X \rangle \ x) {...}
\text{Box}\langle ? \rangle \ m2(\text{Box}\langle ? \rangle \ y) \ { \ this.m1(y); \ }

is translated to (note how opening the existential type allows us to provide an actual type parameter to m1):
\langle X \rangle \text{Box}\langle X \rangle \ m1(\text{Box}\langle X \rangle \ x) {...}
\exists Z.\text{Box}\langle Z \rangle \ m2(\exists Y.\text{Box}\langle Y \rangle \ y) \ {
\text{open } y,Y \ \text{as } y2 \ \text{in}
\text{close}
\text{this.}\langle Y \rangle m1(y2) \ \text{has type Box}\langle Y \rangle \\
\text{with } Z \ \text{hiding } Y; \ \text{has type } \exists Z.\text{Box}\langle Z \rangle 
\}

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

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