Existential Types

Klaus Ostermann
University of Tübingen
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: opened) to use it
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

```latex
let {Counter, counter} = counterADT in
let add3 = \(\lambda c\).\text{Counter}\) counter.inc (counter.inc (counter.inc c)) in
counter.get (add3 counter.new);

\[4 : \text{Nat}\]

let {Counter, counter} = counterADT in

let {FlipFlop, flipflop} = 
\{\text{*Counter,}\n  \{\text{new} = \text{counter.new,}\n   \text{read} = \lambda c:\text{Counter}.\ \text{isSeven} (\text{counter.get} c),\n   \text{toggle} = \lambda c:\text{Counter}.\ \text{counter.inc} c,\n   \text{reset} = \lambda c:\text{Counter}.\ \text{counter.new}\}\}\n  \text{as \{\exists FlipFlop,}\n  \{\text{new:} \text{FlipFlop, read:} \text{FlipFlop} \rightarrow \text{Bool,}\n   \text{toggle:} \text{FlipFlop} \rightarrow \text{FlipFlop, reset:} \text{FlipFlop} \rightarrow \text{FlipFlop}\}\}\in

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

\[\text{false : Bool}\]
```
Existential Types

New syntactic forms

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

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

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

New typing rules

\[ \Gamma \vdash t_1 : T \]
\[ \Gamma \vdash \{\ast U, t_2\} \text{ as } \{\exists X, T_2\} \]
\[ : \{\exists X, T_2\} \]

New evaluation rules

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

\[ (E-UNPACKPACK) \]

\[ \frac{t_{12} \rightarrow t'_{12}}{\{\ast T_{11}, t_{12}\} \text{ as } T_1 \rightarrow \{\ast T_{11}, t'_{12}\} \text{ as } T_1} \]

\[ (E-PACK) \]

\[ \frac{t_1 \rightarrow t'_1}{\text{let } \{X, x\} = t_1 \text{ in } t_2 \rightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2} \]

\[ (E-UNPACK) \]

\[ \frac{\Gamma \vdash t_2 : [X \leftarrow U]T_2 \quad \Gamma \vdash \{\ast U, t_2\} \text{ as } \{\exists X, T_2\}}{\Gamma} \]

\[ (T-PACK) \]

\[ \frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\}}{\Gamma, X, x : T_{12} \vdash t_2 : T_2} \]

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

\[ (T-UNPACK) \]
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

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

- 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

```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>

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

```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
```