Chapter 24: Existential Types

Existential Types
Power of Existential Types
Encoding Existential Types
Two Views of Existential Type \( \{\exists X, T\} \)

- **Logical Intuition**: an element of \( \{\exists X, T\} \) is a value of type \([X \to S]T\), for some type \( S \).

- **Operational Intuition**: an element of \( \{\exists X, T\} \) is a pair, written \( \{S, t\} \), of a type \( S \) and a term \( t \) of type \([X \to S]T\).
  - Like modules and abstract data types found in programming languages.

Example:

\[ p = \{\text{Nat}, \{a=5, f=\lambda x:\text{Nat}. \text{succ}(x)\}\} \]

as \( \{\exists X, \{a:X, f:X \to X\}\} \);
Existential Types

New syntactic forms

\[
\begin{align*}
t & ::= \ldots \\
& \{ \ast T, t \} \text{ as } T \\
& \text{let } \{ X, x \} = t \text{ in } t
\end{align*}
\]

\[
\begin{align*}
v & ::= \ldots \\
& \{ \ast T, v \} \text{ as } T
\end{align*}
\]

\[
\begin{align*}
T & ::= \ldots \\
& \{ \exists X, T \}
\end{align*}
\]

New evaluation rules

\[
\begin{align*}
\text{let } \{ X, x \} = \{ \ast T_{11}, v_{12} \} \text{ as } T_1 & \text{ in } t_2 \\
& \rightarrow [X \rightarrow T_{11}][x \rightarrow v_{12}]t_2
\end{align*}
\]

New terms:

- packing
- unpacking

\[
\begin{align*}
t_{12} & \rightarrow t'_{12} \\
\{ \ast T_{11}, t_{12} \} & \text{ as } T_1 \\
& \rightarrow \{ \ast T_{11}, t'_{12} \} \text{ as } T_1
\end{align*}
\]

Values:

- package value

\[
\begin{align*}
t_1 & \rightarrow t'_1 \\
\text{let } \{ X, x \} = t_1 \text{ in } t_2 \\
& \rightarrow \text{let } \{ X, x \} = t'_1 \text{ in } t_2
\end{align*}
\]

New typing rules:

Type:

\[
\Gamma \vdash t : T
\]

Existential type:

\[
\begin{align*}
\Gamma & \vdash \{ \ast \Gamma, t_2 \} \text{ as } \{ \exists \Gamma, T_2 \} \\
& : \{ \exists \Gamma, T_2 \}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash t_1 : \{ \exists \Gamma, T_{12} \} \\
\Gamma, X, x : T_{12} & \vdash t_2 : T_2 \\
\Gamma & \vdash \text{let } \{ X, x \} = t_1 \text{ in } t_2 : T_2
\end{align*}
\]
Small Examples

• \( p_4 = \{*\text{Nat}, \{a=0, f=\lambda x:\text{Nat}. \text{succ}(x)\}\} \) as \( \exists X, \{a:X, f:X \rightarrow \text{Nat}\}\);
  - \( p_4 : \exists X, \{a:X, f:X \rightarrow \text{Nat}\} \)

• let \( \{X,x\}=p_4 \) in \( (x.f \ x.a) \);
  - \( 1 : \text{Nat} \)

• let \( \{X,x\}=p_4 \) in \( (\lambda y:X. x.f \ y) \ x.a \);
  - \( 1 : \text{Nat} \)

• let \( \{X,x\}=p_4 \) in \( \text{succ}(x.a) \);
  - Error: argument of \( \text{succ} \) is not a number
  - The only operations allowed on \( x \) are those warranted by its “abstract type” \( \{a:X, f:X \rightarrow \text{Nat}\} \)
App1: Data Abstraction with Extentials

- Abstract Data Type

```
ADT counter =
  type Counter
representation Nat
signature
  new : Counter,
  get : Counter → Nat,
  inc : Counter → Counter;
operations
  new = 1,
  get = λi:Nat. i,
  inc = λi:Nat. i+1
```

For external use

Hidden Internal implementation
Abstract Data Type in Existential Types

counterADT =
{*
{new = 1,
get = \lambda i: Nat. i,
inc = \lambda i: Nat. succ(i)}
}
as
{\exists Counter,
{new: Counter,
get: Counter \rightarrow Nat,
inc: Counter \rightarrow Counter}};
• Use Examples

```haskell
let {Counter, counter} = counterADT
  in counter.get (counter.inc counter.new);
⇒ 2 : 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));
```
• Representation-Independent

counterADT =
  \{{x: \text{Nat}},
    \{\text{new} = \{x=1\},
    \text{get} = \lambda i: \{x: \text{Nat}\}. i.x, \\
    \text{inc} = \lambda i: \{x: \text{Nat}\}. \{x=\text{succ}(i.x)\}\} \}
  \exists \text{Counter},
    \{\text{new: Counter, get: Counter→Nat, inc: Counter→Counter}\};

• counterADT : \exists \text{Counter},
  \{\text{new:Counter, get:Counter→Nat, inc:Counter→Counter}\}
App2: Existential Object

\[
c = \{ \text{*Nat,}
\{\text{state = 5,}
\text{methods = \{get = \lambda x: \text{Nat. } x,}
\text{\quad \text{inc = \lambda x: \text{Nat. } succ(x)}\}\}}
\}
\]
as Counter;

where:

\[
\text{Counter = \{ }\exists X, \{ \text{state: } X, \text{methods: } \{ \text{get: } X \to \text{Nat, inc: } X \to X \}\}\}\};
\]

Example:
let \{X, body\} = c in body.methods.get(body.state);
Encoding Existentials

- **Pair can be encoded in System F.**
  \[
  \{U,V\} = \forall X. (U \to V \to X) \to X
  \]

  \[
  \text{pair} : U \to V \to \{U,V\}
  \]
  \[
  \text{pair} = \lambda n1:U. \lambda n2:V. \\
  \quad \lambda X. \lambda f:U \to V \to X. f \ n1 \ n2;
  \]

  \[
  \text{fst} : \{U,V\} \to U
  \]
  \[
  \text{fst} = \lambda p:\{U,V\}. p [U] (\lambda n1:U. \lambda n2:V. n1);
  \]

  \[
  \text{snd} : \{U,V\} \to V
  \]
  \[
  \text{snd} = \lambda p:\{U,V\}. p [V] (\lambda n1:U. \lambda n2:V. n2);
  \]
Existential Encoding

$$\{\exists X, T\} = \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$$

{*S,t} as {∃X,T} = λY. λf:(∀X.T→Y). f [S] t

let {X,x}=t1 in t2 = t1 [T2] (λX. λx:T11.t2)
  (if x :: T11, let ... t2: T2)

Exercise: Show that

let {X,x}={*T11,v12} as T1 in t2
$$\rightarrow[X\mapsto T11][x\mapsto v12]t2$$
Preliminary: syntax, operational semantics

Untyped lambda calculus

Simply typed lambda calculus

Simple extension: tuples/records, sums, lists

Subtyping

Universal type: system F

Reference

Case Study

Typing

FJ

Recursive Types

ADT
Preliminary: syntax, operational semantics

Untyped lambda calculus

Simply typed lambda calculus

Simple extension: tuples/records, sums, lists

Universal type: system F

Reference

Subtyping

Typing

ADT

poly + subtype

Recursive Types

Case Study

FJ

type operator

higher order poly + subtype & kinding
谢谢大家的合作！
欢迎加入北大程序设计语言实验室！