编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2023

Chap 24: Existential Types

Existential Types

Data Abstraction

Encodings in System F

Review: System F

Syntax

$$t ::= \ldots \mid \lambda X.\, t \mid t\ [T] \qquad v ::= \ldots \mid \lambda X.\, t$$

$$T ::= X \mid T \to T \mid \forall X.T \qquad \Gamma ::= \varnothing \mid \Gamma, x : T \mid \Gamma, X$$

Evaluation

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]}\ \text{E-TApp}$$

$$\frac{}{(\lambda X.\, t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12}}\ \text{E-TappTabs}$$

Typing

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.\, t_2 : \forall X.T_2}\ \text{T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}}\ \text{T-TApp}$$

Two Views of a Universal Type $\forall X.T$

Logical Intuition

- An element of $\forall X.T$ is a value of type $[X \mapsto S]T$ **for all** choices of S.
- The identify function $\lambda X.\ \lambda x{:}X.\ x$ erases to $\lambda x.\ x$, mapping a value of any type S to a value of the same type.

Operational Intuition

- An element of $\forall X.T$ is a **function** mapping **any** type S to a specialized term with type $[X \mapsto S]T$.
- In the (E-TAPPTABS) rule, the reduction of a type application is an actual computation step.

Question

We have already seen universal quantifiers $\forall$. What about existential quantifiers $\exists$?

Two Views of an Existential Type $\exists X.T$

Logical Intuition

An element of $\exists X.T$ is a value of type $[X \mapsto S]T$ **for some** type S.

Operational Intuition

An element of $\exists X.T$ is a **pair** of **some** type S and a term of type $[X \mapsto S]T$.

Remark

We will focus on the operational view of existential types.
The essence of existential types is that they **hide information** about the packaged type.

Notations

We write $\{\exists X, T\}$ (instead of $\exists X.T$) to emphasize the operational view.
The pair of type $\{\exists X, T\}$ is written $\{{}^{*}S, t\}$ of a type S and a term t of type $[X \mapsto S]T$.

A Simple Example

Example

The pair

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}}
```

has the existential type $\{\exists X, \{a : X, f : X \to X\}\}$.

- The type component of p is Nat.
- The value component is a record containing of field a of type X and a field f of type $X \to X$, **for some X**.

Example

The same pair p also has the type $\{\exists X, \{a : X, f : X \to \text{Nat}\}\}$.
In general, the typechecker cannot decide **how much information should be hidden**.

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▶ p : {∃X, {a:X, f:X→X}}
p1 = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p1 : {∃X, {a:X, f:X→Nat}}
```

Introduction Rule for $\{\exists X, T\}$

Typing

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{{}^*U, t_2\} \text{ as } \{\exists X, T_2\} : \{\exists X, T_2\}} \text{ T-PACK}$$

Example

Pairs with **different** hidden representation types can inhabit the **same** existential type.

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p4 : {∃X, {a:X, f:X→Nat}}
p5 = {*Bool, {a=ture, f=λx:Bool. if x then 1 else 0}} as {∃X, {a:X, f:X→Nat}};
▶ p5 : {∃X, {a:X, f:X→Nat}}
```

Elimination Rule for $\{\exists X, T\}$

Typing

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

Example

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p4 : {∃X, {a:X, f:X→Nat}}

let {X,x}=p4 in (x.f x.a);
▶ 1 : Nat
let {X,x}=p4 in (λy:X. x.f y) x.a;
▶ 1 : Nat
```

Subtlety of the Elimination Rule

Example

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▶ p4 : {∃X, {a:X, f:X→Nat}}

let {X,x}=p4 in succ(x.a);
▶ Error: argument of succ is not a number

let {X,x}=p4 in x.a;
▶ Error: scoping error!
```

Aside

A simple solution for the scoping problem is to add a well-formedness check as a premise:

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \qquad \Gamma, X, x : T_{12} \vdash t_2 : T_2 \qquad \Gamma \vdash T_2 \text{ type}}{\Gamma \vdash \mathtt{let}\,\{X, x\} = t_1 \,\mathtt{in}\, t_2 : T_2} \text{ T-UNPACK}$$

Existential Types: Syntax and Evaluation

Syntax

$$
\begin{aligned}
\mathtt{t} &::= \ldots \mid \{{}^{*}\mathsf{T}, \mathtt{t}\} \text{ as } \mathsf{T} \mid \mathtt{let}\ \{\mathsf{X}, x\} = \mathtt{t}\ \mathtt{in}\ \mathtt{t} \\
v &::= \ldots \mid \{{}^{*}\mathsf{T}, v\} \text{ as } \mathsf{T} \\
\mathsf{T} &::= \ldots \mid \{\exists \mathsf{X}, \mathsf{T}\}
\end{aligned}
$$

Evaluation

$$
\frac{}{\mathtt{let}\ \{\mathsf{X}, x\} = (\{{}^{*}\mathsf{T}_{11}, v_{12}\} \text{ as } \mathsf{T}_1)\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow [\mathsf{X} \mapsto \mathsf{T}_{11}][x \mapsto v_{12}]\mathtt{t}_2} \quad \text{E-UnpackPack}
$$

$$
\frac{\mathtt{t}_{12} \longrightarrow \mathtt{t}'_{12}}{\{{}^{*}\mathsf{T}_{11}, \mathtt{t}_{12}\} \text{ as } \mathsf{T}_1 \longrightarrow \{{}^{*}\mathsf{T}_{11}, \mathtt{t}'_{12}\} \text{ as } \mathsf{T}_1} \quad \text{E-Pack}
$$

$$
\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\mathtt{let}\ \{\mathsf{X}, x\} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow \mathtt{let}\ \{\mathsf{X}, x\} = \mathtt{t}'_1\ \mathtt{in}\ \mathtt{t}_2} \quad \text{E-Unpack}
$$

Data Abstraction

Abstract Data Types (ADTs)

Definition

An abstract data type (ADT) consists of

- a type name A,
- a concrete representation type T,
- implementations of some operations for creating, querying, and manipulating values of type T, and
- an **abstraction boundary** enclosing the representation and operations.

```
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. succ(i);
```

Translating ADTs to Existentials

```
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
```

ADTs and Modules / Packages

Observation

An element of an existential type can be seen as a **module** or a **package**, in the following sense:

```
let {Counter,counter} = <counter module / counter package> in
<rest of program that uses the module / package>
```

```
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
```

Representation Independence

Observation

We can substitute an alternative implementation of the `Counter` ADT and the program will remain typesafe.

```
counterADT =
   {*{x:Nat},
    {new = {x=1},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
 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
let {FlipFlop,flipflop} = ...
```

Existential Objects

Idea

We choose a **purely functional** style, i.e., when we need to change the object's internal state, we instead build a fresh object.

A counter object consists of (i) a number (**its internal state**) and (ii) a pair of methods (**its external interface**):

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};

c = {*Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
    as Counter;
▶ c : Counter
```

Existential Objects

```
let {X,body} = c in body.methods.get(body.state);
▶ 5 : Nat

sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
▶ sendget : Counter → Nat

let {X,body} = c in body.methods.inc(body.state);
▶ Error: scoping error!

sendinc = λc:Counter.
            let {X,body} = c in
              {*X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▶ sendinc : Counter → Counter
```

ADTs vs. Objects

ADTs

$$\texttt{CounterADT} = \{\exists\,\texttt{Counter},\ \{\texttt{new}:\texttt{Counter},\texttt{get}:\texttt{Counter}\to\texttt{Nat},\texttt{inc}:\texttt{Counter}\to\texttt{Counter}\}\}$$

"The abstract type of counters" refers to the (hidden) type Nat, i.e., simple numbers.

ADTs are usually used in a **pack-and-then-open** manner, leading to a **unique** internal representation type.

Objects

$$\texttt{Counter} = \{\exists\,\texttt{X},\ \{\texttt{state}:\texttt{X},\ \texttt{methods}:\{\texttt{get}:\texttt{X}\to\texttt{Nat},\texttt{inc}:\texttt{X}\to\texttt{X}\}\}\}$$

"The abstract type of counters" refers to the whole package, including the number and the implementations.

Objects are kept closed as long as possible and each object carries its **own** representation type.

Observation

The object style is convenient in the presence of **subtyping** and **inheritance**.

ADTs vs. Objects

Question

What about implementing **binary** operations on the same abstract type?

Let us consider a simple case: we want to implement an equality operation for counters.

ADT Style

```
let {Counter,counter} = counterADT in
let counter_eq = λc1:Counter. λc2.Counter. nat_eq (counter.get c1) (counter.get c2)
in <rest of program>
```

Object Style

```
let counter_eq = λc1:Counter. λc2:Counter.
  let {X1,body1} = c1 in
  let {X2,body2} = c2 in
  nat_eq body1.methods.get(body1.state) body2.methods.get(body2.state);
```

ADTs vs. Objects

Remark

The equality operation can be implemented outside the abstraction boundary.

Let us consider implementing an abstraction for sets of numbers.
The concrete representation is labeled trees and is **not** exposed to the outside.
We'd implement a `union` operation that needs to view the **concrete representation of both** arguments.

ADT Style

$$\texttt{NatSetADT} = \{\exists\,\texttt{NatSet},\ \{\ldots,\ \texttt{union}:\texttt{NatSet}\to\texttt{NatSet}\to\texttt{NatSet}\}\}$$

Object Style

$$\texttt{NatSet} = \{\exists\,\texttt{X},\ \{\texttt{state}:\texttt{X},\ \texttt{methods}:\{\ldots,\ \texttt{union}:\texttt{X}\to\texttt{NatSet}\to\texttt{X}\}\}\}$$

Problems: (i) we need recursive types, and (ii) `union` **cannot access the concrete structure of its 2nd argument**.

ADTs vs. Objects

Question (Exercise 24.2.5)

Why can't we use the type

```
NatSet = {∃X, {state:X, methods:{..., union:X→X→X}}}
```

instead?

Answer

We cannot send a `union` message to a `NatSet` object, with another `NatSet` object as an argument of the message:

```
sendunion = λs1:NatSet. λs2:NatSet.
              let {X1,body1} = s1 in
              let {X2,body2} = s2 in
              ... body1.methods.union body1.state body2.state ...
```

Another explanation: objects allow different internal representations, thus `union:X→X → X` is not safe.

Question

In C++, Java, etc., it's not difficult to implement such a `union` operation. How does that work?

Encodings in System F

Review: Encoding Pairs in System F

PRINCIPLE

Encode typing rules for **destructors** as polymorphic function types.

Elimination Rules for Pairs

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \text{ T-PROJ1} \qquad \frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \text{ T-PROJ2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12} \qquad \Gamma, x : T_{11}, y : T_{12} \vdash t_2 : S}{\Gamma \vdash \texttt{let}\ \{x, y\} = t_1\ \texttt{in}\ t_2 : S} \text{ T-LETPAIR}$$

```
Pair T1 T2 = ∀X. (T1→T2→X) → X
```

Encoding Existentials in System F

The Elimination Rule for Existentials

$$\frac{\Gamma \vdash t_1 : \{\exists X, T\} \qquad \Gamma, X, x : T \vdash t_2 : S}{\Gamma \vdash \mathtt{let}\,\{X, x\} = t_1\ \mathtt{in}\, t_2 : S} \text{ T-UNPACK}$$

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

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

$$\mathbf{let}\ \{X, x\} = t1\ \mathbf{in}\ t2 \stackrel{\text{def}}{=} t1\ [S]\ (\boldsymbol{\lambda} X.\ \boldsymbol{\lambda} x{:}T.\ t2)$$

Homework

Question

Show that under the encodings of existentials in System F, we have the following evaluation relation:

$$\texttt{let}\,\{X, x\} = (\{{}^*T_{11}, v_{12}\}\,\texttt{as}\,T_1)\,\texttt{in}\,t_2 \longrightarrow^* [X \mapsto T_{11}][x \mapsto v_{12}]t_2$$