Existential Types

Sijmen van Bommel & Quinten Kock

based on Types and Programming Languages (Pierce 2002)

Recap: Universal types

- Polymorphism
- $\forall X.T$: value has type $T[X:=S]$ for any type S
- type S is abstract: only known after specialization
- id = $\lambda X.\ \lambda x{:}X.\ x \quad : \quad \forall X.\ X \to X$
- erases to untyped $\lambda x.\ x$
- specializes to $\lambda x{:}S.\ x : S \to S$

Existential types

- {∃X, T}: value has type T[X:=S] for ***some*** type X
- value can be seen as pair {*S, t}: a type S and a term t : T[X:=S]
- type S is hidden: only visible in the definition of t

Type abstraction

Different hidden types, same existential type

p : {∃X, {a:X, f:X->Nat}}

A. p = {*Nat, {a=0, f=\x:Nat. x}}
B. p = {*Bool, {a=False, f=\x:Bool. if x then 1 else 0}}

Type ambiguity

Same value, different existential type

p = {*Nat, {v=0, f=λx:Nat. succ(x)}}

A. p : {∃X, {v:X, f:X->X}}
B. p : {∃X, {v:X, f:X->Nat}}
C. p : {∃X, {v:X, f:Nat->Nat}}
D. p : {∃X, {v:Nat, f:Nat->Nat}}

Packing

packing a value can be done using **as**

{*T,t} as U

where T is some type
t is a term/value
U is an existential type

Examples:

- {*Nat, 42} as {∃X, X}
- {*Bool, true} as {∃X, X}

Unpacking

Unpacking can be done using **let** … **=** … **in** …

let {X,x} = p in v

where p is an existentially typed value
X becomes a type
x becomes a term/value
v is a term/value that may contain x

example:

p = {*Nat, {a: 42, get: λn:Nat.n} as {∃X, {X, X -> Nat}}

let {X,x} = p in x.get(x.a)

evaluates to?

Illegal unpacking

p = {*Nat, {a: 42, get: λn:Nat.n} as { ∃ X, {X, X -> Nat}}

let {X,x} = p in succ(x.a) argument of succ is X, not a number

let {X,x} = p in x.a type X escapes scope

Syntax

terms: ...

| {*T, t} as U — *packing*

| let {X,x}=p in v — *unpacking*

values: ...

| {*T,v} as U — *package value*

types: ...

| {∃X, T} — *existential type*

Evaluation rules

$$\text{let } \{X,x\} = (\{{}^*T,t\} \text{ as } U) \text{ in } e \rightarrow e[X := T,\ x := t] \qquad \text{E-UnpackPack}$$

$$\frac{t1 \rightarrow t2}{\{{}^*T,\ t1\} \text{ as } U \rightarrow \{{}^*T,\ t2\} \text{ as } U} \qquad \text{E-Pack}$$

$$\frac{t1 \rightarrow t2}{\text{let } \{X,x\} = t1 \text{ in } e \rightarrow \text{let } \{X,x\} = t2 \text{ in } e} \qquad \text{E-Unpack}$$

Typing rules

$$\frac{\Gamma \vdash t : T[\ X := U\ ]}{\Gamma \vdash \{{*}U,\ t\} \text{ as } \{\exists X,\ T\} : \{\exists X,\ T\}} \quad \text{T-Pack}$$

$$\frac{\Gamma \vdash t1 : \{\exists X,\ T\} \qquad \Gamma,\ X,\ x : T \vdash y : Y}{\Gamma \vdash \text{let } \{X,\ x\} = t1 \text{ in } y : Y} \quad \text{T-Unpack}$$

Abstract Data Types (ADTs)

- Existential types hide actual representation
- Useful for enforcing abstraction boundaries

```
let {X,x}=p in (λy:X. x.f y) x.a
```

- x.f and x.a are values from p.
- X is abstract for Nat, but:

```
let {X,x}=p in succ(x.a)
```

- is forbidden! We are not allowed to use values of type X as Nat outside of p.
- ADTs are like modules:
  - `let {X,x}=p ↔ import p`

ADT examples

- Counter
  - `counterADT = {*Nat, {new=0, get=λi:Nat. i, inc=λi:Nat. succ(i)}} as {∃C, new:C, get: C->Nat, inc: C->C}`
  - Prevents incorrect use (like dec)
- Associative datatypes
  - abstract over hashmap vs treemap vs ...
  - maintain invariants (e.g. that the tree is in order)
- Rational numbers
  - Floating point vs fixed point vs ratio

Existential Objects

counterObject = {*Nat,

{ state = 5,

methods = {get = λx : Nat . x,

inc = λx : Nat . succ(x) }}}

as

{ ∃ X, {state: x, methods: {get: X → Nat, inc: X → X }}}

let {X,body} = counterObject in body.methods.get(body.state) evaluates to?

functions using counters (as existential objects)

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

sendinc : {∃X, {state:X, ...}} → {∃X, {state:X, ...}}

ADTs vs Objects

usage: counter.get(counter.inc(counter.new))

type: { ∃ C, {get: C-> Nat, ...}}

uses internal representation

set of available functions unextendable

full support for binary operators

usage: sendget (sendinc (counterObject))

type: { ∃ C, {state: C, methods: {get:C->Nat, ...}}}

keeps packaged structure

set of available functions can be extended

limited support for binary operators

modern object oriented languages use a hybrid.

Encoding existential types as universal types (with example)

existential type: `{∃C, {get: C->Nat, ...}}`

universal type: `∀Y. (∀C. {get: C->Nat, ...} -> Y) -> Y`

existential value: `{*Nat, {get=id, ...}}`

universal value:
```
λY.λy:(∀C. {get: C->Nat, ...} -> Y).
          y[Nat]({get=id, ...})
```

existential usage: `let (Counter, counter) = p in v`

universal usage: `p[V](λCounter. λcounter. v)`