Existential Types

Motivation

If *universal* quantifiers are useful in programming, then what about *existential* quantifiers?

Motivation

If *universal* quantifiers are useful in programming, then what about *existential* quantifiers?

Rough intuition:

Terms with universal types are *functions* from types to terms.

Terms with existential types are *pairs* of a type and a term.

Concrete Intuition

Existential types describe simple *modules*:

> An existentially typed value is introduced by pairing a type with a term, written `{*S,t}`. (The star avoids syntactic confusion with ordinary pairs.)
>
> A value `{*S,t}` of type `{∃X,T}` is a module with one (hidden) type component and one term component.

Example: `p` = `{*Nat, {a=5, f=λx:Nat. succ(x)}}`
has type `{∃X, {a:X, f:X→X}}`

The type component of `p` is `Nat`, and the value component is a record containing a field `a` of type `X` and a field `f` of type `X→X`, for some `X` (namely `Nat`).

The same package `p = {*Nat, {a=5, f=λx:Nat. succ(x)}}` *also* has type `{∃X, {a:X, f:X→Nat}}`, since its right-hand component is a record with fields `a` and `f` of type `X` and `X→Nat`, for some `X` (namely `Nat`).

This example shows that there is no automatic ("best") way to guess the type of an existential package. The programmer has to say what is intended.
We re-use the "ascription" notation for this:

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

This gives us the "introduction rule" for existentials:

$$\frac{\Gamma \vdash \mathtt{t}_2 : [\mathtt{X} \mapsto \mathtt{U}]\mathtt{T}_2}{\Gamma \vdash \{*\mathtt{U},\mathtt{t}_2\}\ \mathtt{as}\ \{\exists\mathtt{X},\mathtt{T}_2\} : \{\exists\mathtt{X},\mathtt{T}_2\}} \qquad \text{(T-Pack)}$$

Different representations...

Note that this rule permits packages with *different* hidden types to inhabit the *same* existential type.

Example: `p2 = {*Nat, 0} as {∃X,X}`
`p3 = {*Bool, true} as {∃X,X}`

Different representations...

Note that this rule permits packages with *different* hidden types to inhabit the *same* existential type.

Example: `p2 = {*Nat, 0} as {∃X,X}`
`p3 = {*Bool, true} as {∃X,X}`

More useful example:

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}}
p5 = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}}
```

Exercise...

Here are three more variations on the same theme:

```
p6 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}}
p7 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:Nat→X}}
p8 = {*Nat, {a=0, f=λx:Nat. succ(x)}}
     as {∃X, {a:Nat, f:Nat→Nat}}
```

In what ways are these less useful than p4 and p5?

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}}
p5 = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}}
```

The elimination form for existentials

Intuition: If an existential package is like a module, then eliminating (using) such a package should correspond to "open" or "import."

I.e., we should be able to use the components of the module, but the identity of the type component should be "held abstract."

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\exists\mathtt{X},\mathtt{T}_{12}\} \qquad \Gamma, \mathtt{X}, \mathtt{x}{:}\mathtt{T}_{12} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \text{(T-UNPACK)}$$

Example: if

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

then

```
let {X,x} = p4 in (x.f x.a)
```

has type `Nat` (and evaluates to `1`).

Abstraction

However, if we try to use the `a` component of `p4` as a number, typechecking fails:

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

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

This failure makes good sense, since we saw that another package with the same existential type as `p4` might use `Bool` or anything else as its representation type.

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\exists \mathtt{X}, \mathtt{T}_{12}\} \qquad \Gamma, \mathtt{X}, \mathtt{x}{:}\mathtt{T}_{12} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \{\mathtt{X},\mathtt{x}\}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \quad (\text{T-Unpack})$$

Computation

The computation rule for existentials is also straightforward:

$$\begin{array}{l}\texttt{let \{X,x\}=(\{*T}_{11}\texttt{,v}_{12}\texttt{\} as T}_1\texttt{) in t}_2 \\ \qquad \longrightarrow [\texttt{X} \mapsto \texttt{T}_{11}][\texttt{x} \mapsto \texttt{v}_{12}]\texttt{t}_2\end{array} \qquad \text{(E-UnpackPack)}$$

Example: Abstract Data Types

```
counterADT =
   {*Nat,
    {new = 1,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};
let {Counter,counter} = counterADT in
counter.get (counter.inc counter.new);
```

Representation independence

We can substitute another implementation of counters without affecting the code that uses counters:

```
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}};
```

Cascaded ADTs

We can use the counter ADT to define new ADTs that use counters in their internal representations:

```
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}}

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

Existential Objects

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

Existential objects: invoking methods

More generally, we can define a little function that "sends the `get` message" to any counter:

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

Invoking the `inc` method of a counter object is a little more complicated. If we simply do the same as for `get`, the typechecker complains

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

because the type variable `X` appears free in the type of the body of the `let`.

Indeed, what we've written doesn't make intuitive sense either, since the result of the `inc` method is a bare internal state, not an object.

To satisfy both the typechecker and our informal understanding of what invoking `inc` should do, we must take this fresh internal state and repackage it as a counter object, using the same record of methods and the same internal state type as in the original object:

```
c1 = let {X,body} = c in
       {*X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
     as Counter;
```

More generally, to "send the `inc` message" to a counter, we can write:

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

Objects vs. ADTs

The examples of ADTs and objects that we have seen in the past few slides offer a revealing way to think about the differences between "classical ADTs" and objects.

- Both can be represented using existentials
- With ADTs, each existential package is opened as early as possible (at creation time)
- With objects, the existential package is opened as late as possible (at method invocation time)

These differences in style give rise to the well-known pragmatic differences between ADTs and objects:

- ADTs support binary operations
- objects support multiple representations

A full-blown existential object model

What we've done so far is to give an account of "object-style" encapsulation in terms of existential types.

To give a full model of all the "core OO features" we have discussed before, some significant work is required. In particular, we must add:

- subtyping (and "bounded quantification")
- type operators ("higher-order subtyping")