CIS 500

Software Foundations

Fall 2002

4 December

Announcement

Simon Peyton Jones (Microsoft Research) will be giving a joint CIS / Wharton distinguished lecture tomorrow:

> Composing contracts: an adventure in financial engineering
>
> Thursday, December 5th, 2002
> Huntsman Hall, Room G60
> 3:00 p.m. - 4:30 p.m.

Highly recommended!!

Universal Types

Motivation

In the simply typed lambda-calculus, we often have to write several versions of the same code, differing only in type annotations.

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x)
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x)
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x)
```

This violates a basic principle of software engineering:

Write each piece of functionality once

Motivation

In the simply typed lambda-calculus, we often have to write several versions of the same code, differing only in type annotations.

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x)
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x)
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x)
```

This violates a basic principle of software engineering:

> Write each piece of functionality once... and parameterize it on the details that vary from one instance to another.

Here, the details that vary are the types!

Idea

So we'd like to be able to take a piece of code and "abstract out" some type annotations.

We've already got a mechanism for doing this with terms: λ-abstraction. So let's just re-use the notation for abstracting out types.

Abstraction:

```
double = λX. λf:X→X. λx:X. f (f x)
```

Application:

```
double [Nat]
double [Bool]
```

Computation:

```
double [Nat] ⟶ λf:Nat→Nat. λx:Nat. f (f x)
```

(N.b.: Type application is usually written `t [T]`, though `t T` would be more consistent.)

Idea

What is the type of a term like

$$\lambda X.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ f\ (f\ x)\ ?$$

This term is a function that, when applied to a type X, yields a term of type $(X{\to}X){\to}X{\to}X$.

Idea

What is the *type* of a term like

$$\lambda X.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ f\ (f\ x)\ ?$$

This term is a function that, when applied to a type X, yields a term of type $(X{\to}X){\to}X{\to}X$.

I.e., for all types X, it yields a result of type $(X{\to}X){\to}X{\to}X$.

Idea

What is the **type** of a term like

λ`X. `λ`f:X→X. `λ`x:X. f (f x)` ?

This term is a function that, when applied to a type `X`, yields a term of type `(X→X)→X→X`.

I.e., for all types `X`, it yields a result of type `(X→X)→X→X`.

We'll write it like this: $\forall$`X. (X→X)→X→X`

System F

System F (aka "the polymorphic lambda-calculus") formalizes this idea by extending the simply typed lambda-calculus with type abstraction and type application.

`t ::=`		terms
	`x`	variable
	`λx:T.t`	abstraction
	`t t`	application
	`λX.t`	type abstraction
	`t [T]`	type application

System F

System F (aka "the polymorphic lambda-calculus") formalizes this idea by extending the simply typed lambda-calculus with type abstraction and type application.

`t ::=`		terms
	`x`	variable
	`λx:T.t`	abstraction
	`t t`	application
	`λX.t`	type abstraction
	`t [T]`	type application
`v ::=`		values
	`λx:T.t`	abstraction value
	`λX.t`	type abstraction value

System F: new evaluation rules

$$\boxed{\dfrac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]}} \qquad \text{(E-TApp)}$$

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

System F: Types

To talk about the types of “terms abstracted on types,” we need to introduce a new form of types:

$\mathtt{T}$	::=	types
	$\mathtt{X}$	type variable
	$\mathtt{T}\rightarrow\mathtt{T}$	type of functions
	$\forall\mathtt{X.T}$	universal type

System F: Typing Rules

$$\frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x{:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{x{:}T_1.t_2} : \mathtt{T_1 {\to} T_2}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11} {\to} T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1\ t_2} : \mathtt{T_{12}}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{X.t_2} : \forall \mathtt{X.T_2}} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \forall \mathtt{X.T_{12}}}{\Gamma \vdash \mathtt{t_1\ [T_2]} : [\mathtt{X} \mapsto \mathtt{T_2}]\mathtt{T_{12}}} \qquad \text{(T-TAPP)}$$

Examples

[on board]

Properties of System F

Preservation and Progress. (Proofs similar to what we've seen.)

Strong normalization: every well-typed program halts. (Proof is challenging!)

Type reconstruction: undecidable (major open problem from 1972 until 1994, when Joe Wells solved it)

Parametricity

Observation:

The type $\forall X.\ X \to X \to X$ has exactly two members (up to observational equivalence).

$\forall X.\ X \to X$ has one.

etc.

The concept of parametricity gives rise to some useful "free theorems..."

History

Interestingly, System F was invented independently and almost simultaneously by a computer scientist (John Reynolds) and a logician (Jean-Yves Girard).

Their results look very different at first sight — one is presented as a tiny programming language, the other as a variety of second-order logic.

The similarity (indeed, isomorphism!) between them is an example of the Curry-Howard Correspondence.

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`).

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}}
```

The same package $p = \{*Nat, \{a=5, f=\lambda x:Nat. succ(x)\}\}$ also has type $\{\exists X, \{a:X, f:X\to Nat\}\}$, since its right-hand component is a record with fields a and f of type X and $X\to Nat$, 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:

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

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."

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

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.

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

Computation

The computation rule for existentials is also straightforward:

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

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}} in

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")