CIS 500
Software Foundations
Fall 2006

December 4

Administrivia

Homework 11

Homework 11 is currently due on Friday.

Should we make it due next Monday instead?

More on Evaluation Contexts

Progress for FJ

Theorem [Progress]: Suppose `t` is a closed, well-typed normal form. Then either

1. `t` is a value, or
2. $\mathtt{t} \longrightarrow \mathtt{t}'$ for some $\mathtt{t}'$, or
3. for some evaluation context E, we can express `t` as

$$\mathtt{t} = E[\mathtt{(C)(new\ D(}\overline{\mathtt{v}}\mathtt{))}]$$

with $\mathtt{D} \not<: \mathtt{C}$.

Evaluation Contexts

E ::=	*evaluation contexts*
`[]`	*hole*
E`.f`	*field access*
E`.m(`$\overline{\mathtt{t}}$`)`	*method invocation (rcv)*
`v.m(`$\overline{\mathtt{v}}$`,`E`,`$\overline{\mathtt{t}}$`)`	*method invocation (arg)*
`new C(`$\overline{\mathtt{v}}$`,`E`,`$\overline{\mathtt{t}}$`)`	*object creation (arg)*
`(C)`E	*cast*

E.g.,

```
[].fst
[].fst.snd
new C(new D(), [].fst.snd, new E())
```

Evaluation Contexts

$E[\texttt{t}]$ denotes "the term obtained by filling the hole in E with $\texttt{t}$."

E.g., if $E = \texttt{(A)[]}$, then

$$E[\texttt{(new Pair(new A(), new B())).fst}] \\ = \\ \texttt{(A)((new Pair(new A(), new B())).fst)}$$

Evaluation Contexts

Evaluation contexts capture the notion of the "next subterm to be reduced":

- By ordinary evaluation relation:

  $$\texttt{(A)}\underline{\texttt{((new Pair(new A(), new B())).fst)}} \longrightarrow \texttt{(A)(new A())}$$

  by E-CAST with subderivation E-PROJNEW.

- By evaluation contexts:

  $E = \texttt{(A)[]}$
  $\texttt{r} = \texttt{(new Pair(new A(), new B())).fst}$
  $\texttt{r}' = \texttt{new A()}$
  $\texttt{r} \longrightarrow \texttt{r}'$ by E-PROJNEW
  $E[\texttt{r}] = \texttt{(A)((new Pair(new A(), new B())).fst)}$
  $E[\texttt{r}'] = \texttt{(A)(new A())}$

Precisely...

Claim 1: If $\texttt{r} \longrightarrow \texttt{r}'$ by one of the computation rules E-PROJNEW, E-INVKNEW, or E-CASTNEW and E is an arbitrary evaluation context, then $E[\texttt{r}] \longrightarrow E[\texttt{r}']$ by the ordinary evaluation relation.

Claim 2: If $\texttt{t} \longrightarrow \texttt{t}'$ by the ordinary evaluation relation, then there are unique E, $\texttt{r}$, and $\texttt{r}'$ such that

1. $\texttt{t} = E[\texttt{r}]$,
2. $\texttt{t}' = E[\texttt{r}']$, and
3. $\texttt{r} \longrightarrow \texttt{r}'$ by one of the computation rules E-PROJNEW, E-INVKNEW, or E-CASTNEW.

Proofs: Homework 11.

The Curry-Howard Correspondence

Intro vs. elim forms

An *introduction form* for a given type gives us a way of *constructing* elements of this type.

An *elimination form* for a type gives us a way of *using* elements of this type.

The Curry-Howard Correspondence

In *constructive logics*, a proof of P must provide *evidence* for P.

- "law of the excluded middle"

  $$\overline{P \vee \neg P}$$

  not recognized.
- A proof of $P \wedge Q$ is a *pair* of evidence for P and evidence for Q.
- A proof of $P \supset Q$ is a *procedure* for transforming evidence for P into evidence for Q.

Propositions as Types

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type `P→Q`
proposition $P \wedge Q$	type `P × Q`
proof of proposition P	term `t` of type `P`
proposition P is provable	type `P` is inhabited (by some term)
???	evaluation

Propositions as Types

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type `P→Q`
proposition $P \wedge Q$	type `P × Q`
proof of proposition P	term `t` of type `P`
proposition P is provable	type `P` is inhabited (by some term)
proof simplification (a.k.a. "cut elimination")	evaluation

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)
```

Bad! 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)
```

Bad! 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.

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)
```

Bad! 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

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.

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 commonly written `t [T]`, though `t T` would be more consistent.)

Idea

What is the *type* of a term like

    λX. λf:X→X. λx: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`.

Idea

What is the *type* of a term like

    λX. λf:X→X. λx: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`.

Idea

What is the *type* of a term like

    λX. λf:X→X. λx: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: `∀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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] \longrightarrow \mathtt{t}_1'\ [\mathtt{T}_2]} \quad \text{(E-TApp)}$$

$$(\lambda \mathtt{X}.\mathtt{t}_{12})\ [\mathtt{T}_2] \longrightarrow [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{t}_{12} \quad \text{(E-TappTabs)}$$

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*
	$\mathtt{T}\to\mathtt{T}$	*type of functions*
	$\forall\mathtt{X}.\mathtt{T}$	*universal type*

System F: Typing Rules

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

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t}_2 : \forall \mathtt{X}.\mathtt{T}_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \quad \text{(T-TApp)}$$

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*.

Examples

Lists

```
cons : ∀X. X → List X → List X
head : ∀X. List X → X
tail : ∀X. List X → List X
nil : ∀X. List X
isnil : ∀X. List X → Bool

map =
  λX. λY.
    λf: X→Y.
      (fix (λm: (List X) → (List Y).
              λl: List X.
                if isnil [X] l
                  then nil [Y]
                  else cons [Y] (f (head [X] l))
                                (m (tail [X] l))));

l = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));

head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
```

Church Booleans

```
CBool = ∀X.X→X→X;

tru = λX. λt:X. λf:X. t;
fls = λX. λt:X. λf:X. f;

not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
```

Church Numerals

```
CNat = ∀X. (X→X) → X → X;

c0  =  λX. λs:X→X. λz:X. z;
c1   =  λX. λs:X→X. λz:X. s z;
c2   =  λX. λs:X→X. λz:X. s (s z);

csucc = λn:CNat. λX. λs:X→X. λz:X. s (n [X] s z);

cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
```

Properties of System F

Preservation and Progress: unchanged.

(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: Polymorphic functions cannot do very much with their arguments.

- The type `∀X. X→X→X` has exactly two members (up to observational equivalence).
- `∀X. X→X` has one.
- etc.

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

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:

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

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."

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

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.

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

Computation

The computation rule for existentials is also straightforward:

$$\begin{array}{l}\mathtt{let}\ \{\mathtt{X},\mathtt{x}\}\mathtt{=(\{*T_{11},v_{12}\}\ as\ T_1)}\ \mathtt{in}\ \mathtt{t_2} \\ \qquad \longrightarrow [\mathtt{X} \mapsto \mathtt{T_{11}}][\mathtt{x} \mapsto \mathtt{v_{12}}]\mathtt{t_2}\end{array} \quad \text{(E-UnpackPack)}$$

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")