CIS 500
Software Foundations
Fall 2006

December 6

Administrivia

Administrivia

- No recitations this week
- Extra office hours will be posted to the class mailing list
- Exam: Wednesday, Dec 20, 9–11
  - Location: *Towne 313* (not here!)
  - Coverage: Chapters 1 to 19, 22, and 23 of TAPL, excluding 12 and 15.6, plus reading knowledge of basic OCaml
- Hints: The exam is extremely likely to include...
  - at least one question that is very similar to a homework problem from the past month
  - at least one problem taken verbatim from a one-star exercise in TAPL
  - at least one problem involving proofs
  - at least one problem from chapter 22 and/or 23 (universal and existential types) — but nothing too complicated

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:

$$\frac{\Gamma \vdash \mathtt{t}_2 : [\mathtt{X} \mapsto \mathtt{U}]\mathtt{T}_2}{\Gamma \vdash \mathtt{\{*U,t_2\}\ as\ \{\exists X,T_2\}} : \mathtt{\{\exists X,T_2\}}} \quad \text{(T-Pack)}$$

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."

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{\{\exists X,T_{12}\}} \qquad \Gamma, \mathtt{X}, \mathtt{x}{:}\mathtt{T}_{12} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ \{X,x\}=t_1\ in\ t_2} : \mathtt{T}_2} \quad \text{(T-Unpack)}$$

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:

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

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")

Recap... Where we've been

What is "software foundations"?

Software foundations (a.k.a. "theory of programming languages") is the study of the *meaning* of programs.

A main goal is finding ways to describe program behaviors that are both *precise* and *abstract*.

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)
  Important in some domains (safety-critical systems, hardware design, inner loops of key algorithms, ...), but currently very difficult and expensive. We have not said much about this in the course.
- To develop intuitions for informal reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or security properties)
- To understand language features (and their interactions) deeply and develop principles for better language design

PL as the "materials science" of computer science...

What I hope you got out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - How to view programs and whole languages as formal, mathematical objects
  - How to make and prove rigorous claims about them
  - Detailed study of a range of basic language features
- Deep intuitions about key language properties such as type safety
- Familiarity with today's best tools for language design, description, and analysis

Programming languages are everywhere. Most software designers are — at some point — language designers!

Overview

In this course, we concentrated on operational semantics and type systems.

- Part O: Background
  - A taste of OCaml
  - Functional programming style
- Part I: Basics
  - Inductive definitions and proofs
  - Operational semantics
  - The lambda-calculus
  - Evaluator implementation in OCaml

- Part II: Type systems
  - Simple types
  - Type safety — preservation and progress
  - Formal description of a variety of basic language features (records, variants, lists, casting, ...)
  - References
  - Exceptions
  - Subtyping
  - Metatheory of subtyping (subtyping and typechecking algorithms)
  - Polymorphism (universal and existential types)
- Part III: Object-oriented features (case studies)
  - A simple imperative object model
  - An direct formalization of core Java

What next?

The rest of TAPL

Several more core topics are covered in the second half of TAPL.

- Recursive types (including a rigorous treatment of induction and co-induction)
- More on parametric polymorphism (universal and existential types)
  - Bounded quantification
  - Refinement of the imperative object model
  - ML-style type inference
- Type operators
  - Higher-order bounded quantification
  - A purely functional object model

The Research Literature

With this course under your belt, you are ready to directly address research papers in programming languages.

This is a big area, and each sub-area has its own special techniques and notations, but you now have pretty much all the basic intuitions needed to understand these on your own.

The End