CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 37
Existential Types and Modules
22 Nov 00

Existential types

- Last time: *existential* types $\exists X.\sigma$
- Existential type hides some part of a type
- Value is a pair $[\tau, v]$ where τ is the hidden part
  - Intuition: $[\tau, v] : \exists X.\sigma$ if $v : \sigma\{\tau/X\}$
- Creation:

$$\frac{\Delta;\Gamma \vdash e\{\tau/X\} : \sigma\{\tau/X\}}{\Delta;\Gamma \vdash \text{pack } [X = \tau,\ e] : \exists X.\sigma}$$

pack [bool, ⟨#t, λx:bool.x⟩] : $\exists X.X*(X\to\text{bool})$
pack [int, ⟨5, λx:int.#t⟩] : $\exists X.X*(X\to\text{bool})$

Elimination

- Existential value used via unpack

unpack e_1 as $[X, x]$ in e_2

  - Bind components of existential value e1 to type variable X, variable x
  - X must be fresh ($X \notin \Delta$), cannot escape from unpack ($\Delta \vdash \sigma_2$)

$$\frac{\Delta;\Gamma \vdash e_1 : \exists Y.\sigma_1 \quad \Delta, X ; \Gamma, x{:}\sigma_1\{X/Y\} \vdash e_2 : \sigma_2 \quad X \notin \Delta \quad \Delta \vdash \sigma_2}{\Delta;\Gamma \vdash \text{unpack } e_1 \text{ as } [X, x] \text{ in } e_2 : \sigma_2}$$

let p : $\exists X.X*X\to\text{bool}$ = pack [int, ⟨5, λx:int.#t⟩] in
unpack p as [X, v] in $(\pi_2 v)(\pi_1 v)$: bool

Operational semantics

- Like fold/unfold, type abstraction/projection: no computational content

unpack (pack $[X{=}\tau,\ v]$) as $[X, x]$ in $e' \to e'\{\tau/X, e/x\}$

v ::= … | pack $[X{=}\tau,\ v]$
C ::= … | pack $[X{=}\tau,\ C]$ | unpack C as $[X, x]$ in e

Modeling Objects

- Can combine with recursion to obtain quasi-objects (sans subtyping, inheritance)

```
class intset {
  public intset union(intset);
  public boolean contains(int);
  private int value;
  private intset+1 left, right;
}
```

⇒

```
intset = μT.∃P.{
  union: T→T,
  contains: int→boolean,
  priv: P
}
```

```
fold_intset pack[ P={ value: int, left: intset, right: intset},
  rec s {priv = {value = 5, left = ..., right = ...},
    contains = λx:int.if x=s.priv.value then #t else
      if x<s.priv.value then
        unpack (unfold s.priv.left) as [S,l] in
          l.contains(x) ... } ]
```


Modules

- (Weak) existential types can't provide full functionality of a module
- **module**—collection of related values *and types*: mechanism for separate compilation, encapsulation, abstraction
  - vs. **record**—set of named fields with types
  - similar: *interface* defines type of module *value*
  - Classic ADTs!

```
IntSet = {
  type T;          (abstract type hides impl.)
  val contains: T*int→bool,
  val union: T*T→T
}
```

```
treeIntSet : IntSet = {
  type T = μX.{value: int,
        left: X+1, right: X+1},
  contains = λs:T,x:int.
    if x < (unfold s).value then
        (unfold s)
  union = λs1,s2:T. ...
}
```

Modules as existentials?

```
IntSet = {
  type T;
  val createEmpty: unit→T,
  val createSingle: int→T,
  val contains: T*int→bool,
  val union: T*T→T }
```

Sets can't escape unpack!

unpack e_1 as $[T_1, m_1]$ in
  unpack e_2 as $[T_2, m_2]$ in
  …
    unpack e_n as $[T_n, m_n]$
in e

IntSet = $\exists T$.{
  createSingle: int→T
  contains: T*int→bool,
  union: T*T→T
}
treeIntSet: IntSet
unpack treeIntSet as [T, m] in … m.union(t1, t2)

Module types, terms

τ ::= … | { type $X_1,\ldots,X_m$; val $l_1{:}\tau_1 \ldots l_n{:}\tau_n$ }
  | $e.X$

e ::= … | { type $X_1{=}\tau_1,\ldots X_m{=}\tau_m$; val $l_1{=}e_1,\ldots l_n{=}e_n$ }
  | $e.l$

```
IntSet = {
  type T;
  val createEmpty: unit→T,
  val createSingle: int→T,
  val contains: T*int→bool,
  val union: T*T→T }
```

```
let treeIntSet: IntSet = … in
let t1:treeIntSet.T = treeIntSet.createSingle(1) in
let t2:treeIntSet.T = treeIntSet.createSingle(2) in
treeIntSet.contains(treeIntSet.union(t1,t2), 0)
```


Modules vs. existentials

τ ::= … | { type $X_1,\ldots,X_m$; val $l_1{:}\tau_1 \ldots l_n{:}\tau_n$ }
  | $e.X$

e ::= … | { type $X_1{=}\tau_1,\ldots X_m{=}\tau_m$; val $l_1{=}e_1,\ldots l_n{=}e_n$ }
  | $e.l$

- Module looks like record, but can define types X_i
- Looks like an existential, but
  - Can define multiple types, multiple values (trivial)
  - Selection expression $e.l$ instead of unpack
  - The types X_i can be mentioned outside the module!
  - *strong existential type*

Strong existential types

τ ::= … | $\exists X.\tau$ | e.T

e ::= … | pack $[X{=}\tau, e]$
  | unpack e_1 as $[X, x]$ in e_2 | e.V

$$\frac{\Delta;\Gamma \vdash e\{\tau/X\} : \sigma\{\tau/X\}}{\Delta;\Gamma \vdash \text{pack } [X{=}\tau,\ e] : \exists X.\sigma} \quad \sout{\Delta \vdash \sigma_2}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \exists Y.\sigma_1 \quad \Delta,X\,;\,\Gamma,x{:}\sigma_1\{X/Y\} \vdash e_2 : \sigma_2 \quad X \notin \Delta}{\Delta;\Gamma \vdash \text{unpack } e_1 \text{ as } [X, x] \text{ in } e_2 : \sigma_2\{e_1.\text{T}/X\}}$$

$$\frac{\Delta;\Gamma \vdash e : \exists X.\sigma \qquad X \notin \Delta}{\Delta;\Gamma \vdash e.\text{V} : \sigma\{e.\text{T}/X\}}$$

Intset module example

IntSet = $\exists T$.{
  createSingle: int→T
  contains: T*int→bool,
  union: T*T→T
}

```
treeIntSet : IntSet = pack [
  T = μX.{ value: int, left, right: [full: X, empty: 1]},
  { createSingle = λx:int.fold_S {value=x,
                              left=#u, right=#u },
   contains = rec c:T*int→bool.λs,v.
     if v = (unfold s).value then #t
     else if v < (unfold s).value then
       case (unfold s).left of
         b(t) ⇒ c(t,v) | lf(u) ⇒ #f
     else if v > …
}]
```


Dependent module types

- Modules, strong existentials: e.T is a type that depends on a value (*dependent type*)
- Must make sure that value of e can't change — $(!x)$.T where x : ref $\exists X.\sigma$ won't be guaranteed to be same type everywhere
- Simple approach: restrict to x.T
  - More conditions: one $x : \exists X.\sigma$ in Γ at a time, x.T can't escape scope of x
  - Most module languages:
    - only x.T; x can only refer to top-level module terms
    - one module value per module type
    - no new module values can be created at run time

First-class vs Second-class

- Second-class modules: linker can providesmodule value for each module type. Type of module used as name for module value: `IntSet.contains, IntSet.T`
- First-class modules: must name module values *explicitly* rather than using name of signature: `treeIntSet.contains, treeIntSet.T`
- Code written to use ADT must be passed module value too! (tells which code to invoke)

```
int contains0(IntSet set, set.T s) {
      return set.contains(s, 0);
}
```