Existential types
- Last time: existential types $\exists X.\sigma$
- Existential type hides some part of a type
- Value is a pair $[\tau, v]$ where $\tau$ is the hidden part
  - Intuition: $[\tau, v]$ : $\exists X.\sigma$ if $v : \sigma(\tau/X)$
- Creation:
  $\Delta;\Gamma \vdash e(\tau/X) : \alpha(\tau/X)$
  $\Delta;\Gamma \vdash \text{pack}[X = \tau, e] : \exists X.\sigma$

pack $[\text{bool}, (#t, \lambda x:\text{bool}.x)] : \exists X.X \rightarrow \text{bool}$
pack $[\text{int}, (5, \lambda x:\text{int}.#t)] : \exists X.X \rightarrow \text{bool}$

Elimination
- Existential value used via unpack
  unpack $e_1$ as $[X, x]$ in $e_2$
  - Bind components of existential value $e_1$ to type variable $X$, variable $x$
  - $X$ must be fresh ($X \not\in \Delta$), cannot escape from unpack ($\Delta/G60\sigma_2$)
  $\Delta;\Gamma \vdash \text{pack}[X = \tau, e] : \exists X.\sigma$

let $p : \exists X.X \rightarrow \text{bool} = \text{pack}[\text{int}, (5, \lambda x:\text{int}.#t)]$
in unpack $p$ as $[X, v]$ in $(\pi_2 v)(\pi_1 v) : \text{bool}$

Operational semantics
- Like fold/unfold, type abstraction/projection: no computational content
unpack (pack $[X=\tau, v]$) as $[X, x]$ in $e' \rightarrow e'(\tau/X, e/x)$

$\mu \tau.∃P.\{\text{union}: \tau \rightarrow \tau, \text{contains}: \tau \rightarrow \text{boolean}, \text{priv}: P\}$

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!

foldintset pack $\{\text{value}: \text{int}, \text{left}: \text{intset}, \text{right}: \text{intset}\}$
rec $s \{\text{priv} = \{\text{value}: 5, \text{left} = \ldots, \text{right} = \ldots\},$
contains $= \lambda x:\text{int}.\text{if } x < (\text{unfold } s).\text{value} \text{ then } #t \text{ else}$
unpack (unfold s).priv.value then $\text{false} \text{ else }$
unpack (unfold s).priv.left) as $[S, l]$ in $l.\text{contains}(x) \ldots$
Modules as existentials?

\[
\text{IntSet} = \{ \\
\text{type } T, \\
\text{val } \text{createEmpty: } \text{unit} \rightarrow T, \\
\text{val } \text{createSingle: } \text{int} \rightarrow T, \\
\text{val } \text{contains: } T \rightarrow \text{bool}, \\
\text{val } \text{union: } T \rightarrow T \}
\]

\[
\text{let } \text{treeIntSet: } \text{IntSet} \text{ in } \\
\text{unpack } \text{treeIntSet} \text{ as } [T, m] \text{ in } \\
\text{m.union(t1, t2)}
\]

Module types, terms

\[
\tau ::= \ldots | \{ \text{type } X_1, \ldots, X_n; \text{val } l_1; \tau_1 \ldots l_n; \tau_n \}
\]

\[
e ::= \ldots | \{ \text{type } X_1=e_1, \ldots, X_m=e_m; \text{val } l_1=e_1, \ldots, l_n=e_n \}
\]

- Module looks like record, but can define types \(X_i\)
- Looks like an existential, but
  - Can define multiple types, multiple values (trivial)
  - Selection expression \(x: e\) instead of unpack
  - The types \(X_i\) can be mentioned outside the module!
- \text{strong existential type}

Strong existential types

\[
\Delta, \Gamma \vdash e : \exists X, \sigma \\
\Delta, \Gamma \vdash \text{unpack } e \text{ as } [X, x] \text{ in } e_2 : e.V
\]

\[
\Delta, \Gamma \vdash e : \exists X, \sigma \\
\Delta, \Gamma \vdash e : e.T/X
\]

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 \(x : \text{ref } \exists X, \sigma\) won't be guaranteed to be same type everywhere
- Simple approach: restrict to \(x.T\)
- More conditions: one \(x : \exists X, \sigma\) in \(\Gamma\) 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

\[
\text{Intset module example}
\]

\[
\text{let } \text{treeIntSet: } \text{IntSet} \text{ in } \\
\text{unpack } \text{treeIntSet} \text{ as } [T, m] \text{ in } \\
\text{m.union(t1, t2)}
\]
First-class vs Second-class

- Second-class modules: linker can provide module 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)

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