CS611 Lecture 39 — Module Types — 3 December 2001

Scribe: Yan Zhang and Antonio Montalban — Lecturer: Andrew Myers

## 1 Review of existential types

In the previous class we defined the existential type $\exists X.\sigma$, where $X$ is a type variable that may appears free $\sigma$. We have the constructor $\mathsf{pack}_{\exists X.\sigma}[\tau, e]$, and the destructor $\mathsf{unpack}\ e_1\ \mathsf{as}\ [Y, x]\ \mathsf{in}\ e_2$. The new rule is

$$\mathsf{unpack}\ (\mathsf{pack}_{\exists X.\sigma}[\tau, v])\ \mathsf{as}\ [Y, \tau]\ \mathsf{in}\ e_2 \mapsto e_2\{\tau/Y, v/x\}$$

Observe that the rule has no computational content. We could erase the types and get the same results.

## 2 Typing Rules

We extend the definition of being a well-formed type, $\Delta \vdash \sigma$, where $\Delta$ is a set of variables, by adding a new rule:

$$\frac{\Delta, X \vdash \tau}{\Delta \vdash \exists X.\tau}$$

And we extend the definition of being a well-typed term, $\Delta; \Gamma \vdash e : \tau$, where $\Gamma \in \mathit{Var} \rightharpoonup \mathit{Type}$, by adding two new rules:

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

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

These correspond, through the Curry-Howard isomorphism, with the followings rules of logic:

$$\frac{\Delta; \Gamma \vdash \phi\{A/X\} \quad \Delta \vdash A \in S}{\Delta; \Gamma \vdash \exists X \in S.\phi}\ \exists\ \textit{introduction}$$

$$\frac{\Delta; \Gamma \vdash \exists X \in S.\phi_1 \quad \Delta, Y; \Gamma, \phi_1\{Y/X\} \vdash \phi_2 \quad \Delta \vdash \phi_2 \quad Y \notin \Delta}{\Delta; \Gamma \vdash \phi_2}\ \exists\ \textit{elimination}$$

## 3 Module Types

One thing we can do with the existential types is to model modules. First we define an extension of $\lambda^{\rightarrow}$ that supports modules. We extend the definition of types as follows:

$$\tau ::= ...\ \mid\ \mathsf{interface}\ \{\mathsf{type}\ X_1, ..., X_m;\ \mathsf{val}\ x_1 : \tau_1, ..., x_n : \tau_n\}\ \mid\ e.X$$

The interface type is not as the Java's interface, it is like the Modula 3's interface and like what in ML is called sig. The types $X_1, ..., X_m$ are *abstract types*; we don't know the actual identities of them. The type $e.X$ is a *dependent type*, because it depends on a term, so its value is decided at runtime.

We extend the set of expressions as follows

$$e ::= ...\ \mid\ \mathsf{module}\ \{\mathsf{type}\ X_1 = \tau_1, ..., X_m = \tau_m;\ \mathsf{val}\ x_1 = e_1, ..., x_n = e_n\}\ \mid\ e.x$$

Now we can abstract data types in a satisfactory way.

## 4 Examples

We define a module that implements rationals:

$$\begin{aligned}&\mathsf{ratint} \equiv \mathsf{interface}\{\mathsf{type}\ T;\\ &\qquad\qquad \mathsf{val\ create} : \mathsf{int} * \mathsf{int} \to T\\ &\qquad\qquad\quad \mathsf{add} : T * T \to T\}\end{aligned}$$

$$\begin{aligned}&\mathsf{let\ ratmod} = \mathsf{module}\{\mathsf{type{=}int} * \mathsf{int};\\ &\qquad \mathsf{val\ create} = \lambda p, q : \mathsf{int}.\langle p, q\rangle,\\ &\qquad\quad \mathsf{add} = \lambda r_1, r_2 : T.\langle \mathsf{left}\ r_1 * \mathsf{right}\ r_2 + \mathsf{left}\ r_2 * \mathsf{right}\ r_1, \mathsf{right}\ r_1 * \mathsf{right}\ r_2\}\\ &\mathsf{in\ ratmod.add}\ (\mathsf{ratmod.create}\langle 1, 2\rangle)(\mathsf{ratmod.create}\langle 3, 4\rangle)\\ &: \mathsf{ratmod.T}\end{aligned}$$

We could have defined some other functions inside the module without exporting them. For example we could have defined a function gcd, that returns the greater common divisor of two numbers, and used it in the definition of add.

Now, we consider the same example but using existential type instead of module types.

$$\begin{aligned}&\mathsf{ratint} \equiv \exists T.\{\mathsf{create} : \mathsf{int} * \mathsf{int} \to T,\\ &\qquad\quad \mathsf{add} : T * T \to T\}\end{aligned}$$

$$\begin{aligned}&\mathsf{let\ ratmod} = \mathsf{pack}_{\mathsf{ratint}}[\mathsf{int} * \mathsf{int}, \{\mathsf{create} = ...,\\ &\qquad\qquad\qquad \mathsf{add} = ...\}]\\ &\mathsf{in\ unpack\ ratmod\ \ as}\ [\mathsf{ratmod\_}T,\ \mathsf{ratmod\_}V]\ \mathsf{in}\ \langle program\rangle\end{aligned}$$

Consider another example from last time:

```
class intset{
   intset union (intsets);
   bool contains (int);
   intset left, right;
   int val;
}
```

Use existential type language, this becomes

$$\begin{aligned}&\mathsf{intset} \equiv \mu S.\exists P\}\mathsf{union} : S \to S,\\ &\qquad\quad \mathsf{contains}\ \ : \mathsf{int} \to \mathsf{bool},\\ &\qquad\quad \mathsf{fields} : P\\ &\qquad\quad \}\end{aligned}$$

We can create an intset as follows:

$$\begin{aligned}&\mathsf{fold}_{\mathsf{intset}}\mathsf{pack}_{\exists P.\{\mathsf{union}:S\to S,\mathsf{contains}:\mathsf{int}\to\mathsf{bool},\mathsf{fields}:P\}}\mathsf{rec\ this}.\{\mathsf{fields} = \{\mathsf{left} = .., \mathsf{right} = .. : \mathsf{intset}, \mathsf{val} = .. : \mathsf{int}\},\\ &\qquad \mathsf{contains} = ...\mathsf{this.fields.left}...,\\ &\qquad \mathsf{union} = \lambda s : \mathsf{intset}.\\ &\qquad\qquad \mathsf{unpack}\ (\mathsf{unfold}\ s)\ \mathsf{as}\ [P', s']\ \mathsf{in}...\}\end{aligned}$$

There is a problem when we try to define union. The problem is that union receives $s$ of type intset, but we don't know how the implementation of intset is in the definition of $s$, so we cannot access what is inside fields.

To solve this problem we can use strong existential types.

## 5 Strong Existential Types

We have been looking at weak existential types. We can extend it as strong existential types by adding terms as:

$$\sigma ::= ... \mid \exists X.\sigma \mid e.X$$

$$e ::= ... \mid \mathsf{pack}_{\exists X,\sigma}[\tau, e] \mid \mathsf{unpack}\ e_1\ as\ [Y, x]\ in\ e_2 \mid e.V$$

Here the term $e.X$ is called *dependent type*, as the one we had with Module Types.

Now the judgment that asserts that a type is well-formed has to have the form

$$\Delta; \Gamma \vdash \sigma$$

because $\sigma$ may depend on a term.

The inference rule for this term is:

$$\frac{\Delta, X; \Gamma \vdash \sigma \quad \Delta; \Gamma \vdash e : \exists X.\sigma}{\Delta; \Gamma \vdash e.X}$$

Observe that to check that $e.X$ is well-formed we have to type-check the expression $e$.

The rule for pack remains same as before, but the rule for unpack is changed as:

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

Here we don't need $\Delta \vdash \sigma_2$ since the hidden type can be talked about with the extended typing rules.

The problem we have now is that if we implement some type in two different ways they are considered different types.

Let's consider the following example:

```
let $p_1 = \mathsf{pack}_{\exists X.X*(X\rightarrow\mathsf{bool})}[\mathsf{int}, \langle 2, \lambda n.n = 2\rangle]$ in
  let $p_2 = \mathsf{pack}_{\exists X.X*(X\rightarrow\mathsf{bool})}[\mathsf{bool}, \langle \#t, \lambda b.b\rangle]$ in
    let $v$ = unpack $p_1$ as $[Y, x]$ in left $x$
    in $f$ = unpack $p_2$ as $[Y, x]$ in right $x$
    in $f\ v$
```

In weak existential types, we are not allowed to write like this. This is because we don't know anything about the hidden type. But if we introduce the strong existential types (which are called generalized sum types, as in Mitchell), we have the dependent type term $e.X$, in this example, which corresponds to $p_1.X$ and $p_2.X$. After substituting $Y$ by $p_1.X$ and $p_2.X$ respectively and type-checking, we find out $f$ has type $p_2.X \rightarrow \mathsf{bool}$ but $v$ has type $p_1.X$. So the code above is wrong since it doesn't type checks right. But if we change the code as:

```
let $p_1 = \mathsf{pack}_{\exists X.X*(X\rightarrow\mathsf{bool})}[\mathsf{int}, \langle 2, \lambda n.n = 2\rangle]$ in
  let $p_2 = \mathsf{pack}_{\exists X.X*(X\rightarrow\mathsf{bool})}[\mathsf{bool}, \langle \#t, \lambda b.b\rangle]$ in
    let $v$ = unpack $p_1$ as $[Y, x]$ in left $x$
    in $f$ = unpack $p_1$ as $[Y, x]$ in right $x$
    in $f\ v$
```

Now $f$ has new type $p_1.X \rightarrow \mathsf{bool}$ and $v$ has type $p_1.X$, so the above code is valid under type-checking. But we notice this only works under strong existential types, for weak existential still doesn't allow us to do this.