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 $\text{pack}_{\exists X.\sigma}[\tau, e]$, and the destructor $\text{unpack} e_1$ as $[Y, x]$ in $e_2$. The new rule is

$$\text{unpack} (\text{pack}_{\exists X.\sigma}[\tau, v]) \Rightarrow [Y, \tau] \text{ in } e_2$$

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:

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

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

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

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

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

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

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

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:

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

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 $X_1, \ldots, X_m$ 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

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

Now we can abstract data types in a satisfactory way.
4 Examples

We define a module that implements rationals:

\[
\text{rat} \equiv \text{interface}\{ \text{type } T; \\
\quad \text{val create} : \text{int} \times \text{int} \rightarrow T \\
\quad \text{add} : T \times T \rightarrow T \}
\]

let ratmod = module\{\text{type }\text{int} \times \text{int}; \\
\quad \text{val create} = \lambda p, q : \text{int}. \langle p, q \rangle, \\
\quad \text{add} = \lambda r_1, r_2 : T. \langle \text{left } r_1 \times \text{right } r_2 + \text{left } r_2 \times \text{right } r_1, \text{right } r_1 \times \text{right } r_2 \rangle \\
\} in \text{ratmod}\.\text{add}\ (\text{ratmod}\.\text{create}(1, 2)) (\text{ratmod}\.\text{create}(3, 4)) : \text{ratmod}\.T

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

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

\[
\text{rat} \equiv \exists T.\{ \text{create} : \text{int} \times \text{int} \rightarrow T, \\
\quad \text{add} : T \times T \rightarrow T \}
\]

let ratmod = \text{pack}_\text{rat} [\text{int} \times \text{int}. \{ \text{create} = \ldots, \\
\quad \text{add} = \ldots \} ] \\
\text{in } \text{unpack } \text{ratmod as } [\text{ratmod}_T, \text{ratmod}_V] \text{ in } \langle \text{program} \rangle

Consider another example from last time:

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

Use existential type language, this becomes

\[
\text{intset} \equiv \mu S.\exists P \{ \text{union} : S \rightarrow S, \\
\quad \text{contains} : \text{int} \rightarrow \text{bool}, \\
\quad \text{fields} : P \}
\]

We can create an intset as follows:

\[
\text{fold}_\text{intset}_\text{pack}\exists P. \{ \text{union} : S \rightarrow S, \text{contains} : \text{int} \rightarrow \text{bool}, \text{fields} : P \} \text{rec this. } \{ \text{fields} = \{ \text{left} = \ldots, \text{right} = \ldots : \text{intset}, \text{val} = \ldots : \text{int} \}, \\
\quad \text{contains} = \ldots \text{this}.\text{fields}.\text{left}, \ldots, \\
\quad \text{union} = \lambda s : \text{intset}. \\
\quad \text{unpack} (\text{unfold } s) \text{ as } [P', s'] \text{ in } \ldots
\]

There is a problem when we try to define \text{union}. The problem is that \text{union} receives \text{s} of type \text{intset}, but we don’t know how the implementation of \text{intset} is in the definition of \text{s}, so we cannot access what is inside \text{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:

\[ \sigma ::= \ldots \mid \exists X.\sigma \mid e.X \]

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

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:

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

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

The rule for \text{pack} remains same as before, but the rule for \text{unpack} is changed as:

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

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:

```haskell
let p1 = pack_{\exists X.\sigma_1[X \rightarrow \text{bool}]\}[\text{int}, (2, \lambda n.n = 2)] \ in
let p2 = pack_{\exists X.\sigma_1[X \rightarrow \text{bool}]\}[\text{bool}, \ldots \}
    let v = unpack p1 as [Y, x] \ in \ \text{left} \ x
    in \ f = \text{unpack} \ p2 \ as \ [Y, x] \ \text{in} \ \text{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 \( p1.X \) and \( p2.X \). After substituting \( Y \) by \( p1.X \) and \( p2.X \) respectively and type-checking, we find out \( f \) has type \( p2.X \rightarrow \text{bool} \) but \( v \) has type \( p1.X \). So the code above is wrong since it doesn’t type checks right. But if we change the code as:

```haskell
let p1 = pack_{\exists X.\sigma_1[X \rightarrow \text{bool}]\}[\text{int}, (2, \lambda n.n = 2)] \ in
let p2 = pack_{\exists X.\sigma _1[X \rightarrow \text{bool}]\}[\text{bool}, \ldots \}
    let v = unpack p1 as [Y, x] \ in \ \text{left} \ x
    in \ f = \text{unpack} \ p1 \ as \ [Y, x] \ \text{in} \ \text{right} \ x
    in \ f \ v
```

Now \( f \) has new type \( p1.X \rightarrow \text{bool} \) and \( v \) has type \( p1.X \), 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.