CS611 Lecture 34 Modules 11/19/04
Scribe: Liviu Popescu Lecturer: Andrew Myers

1 Typing Rules

To support existential types, we must extend the definition of a well-formed type, $\Delta \vdash \sigma$, where Δ is a set of type variables. The new rule is the following:

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

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 correspondence, to the following rules of logic:

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

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

Examples

Consider a sorting procedure written in C. Its declaration will look something like the following:

```
sort(T x[], int compare (T,T)) { ... }
```

This procedure takes as in put a vector x of type T and a previously defined function compare which can compare two elements of type T. The problem with this declaration is that the function compare will not be able to refer to data declared outside the function – except global data, which is both limited and dangerous, especially in a concurrent environment. A better strategy is for sort to take a closure as an argument: a pointer to the compare code along with a pointer to an environment that compare can use. The closure might have the following type:

```
struct compare_closure {
       int compare (T x, T y, void* env) ;
       void* env }
```

Notice here the pointer env. The idea is that when calling a closure c, we write the expression c.compare(x, y, c.env), allowing the comparison function to access resources through env. Recall that the closure conversion that you already implemented generated an environment like this for free variables of a function, representing the values of the free variables at the point where the function is defined. The problem is that different callers to sort may want to provide different comparison functions needing different kinds of environments. No one environment type can be correct, so C programmers use the void * type instead—basically, getting around the type system. A better solution if your language supports it is to use existential types instead, e.g.:

```
∃α.struct compare_closure {
         int compare (T x, T y, α env) ;
         α env }
```

The calling code above is still type-safe, because the caller can see that env and the corresponding argument to compare have the same type α. Yet the sort function does not need to know what α is. The caller can use pack to convert its closure into this more generic form. Existentials of this sort are supported by Cyclone, a safe version of C developed at Cornell; they are useful for precisely this kind of code.

Consider our intset example again. Using existential types, this can be implemented with information hiding. The type intset comprises three fields: "left", "right" and "value", and two operations: "union" and "contains".

Fields are defined as:

$\mathsf{fields} = [\mathsf{left} : \mathsf{intset}, \mathsf{right} : \mathsf{intset}, \mathsf{value} : \mathsf{int}]$

$\mathsf{intset} \triangleq \mu S.\exists P.\{\mathsf{union} : S \rightarrow S, \mathsf{contains} : \mathsf{int} \rightarrow \mathsf{bool}\}$

We can create an intset as follows:

```
fold_intset pack_{∃P.(1+P*{union:S→S,contains:int→bool,}} rec this.{fields * union...
                              {left = empty, right = empty, val = 0}
                              union =λs : intset.
                                    unpack (unfold s) as [P', s'] in...}
                              contains = λn : int.if n = (#1 this).value then true
                                    else if n < this.value then case (#1 this).left of
                                    λu : 1.false
                                    λs' : intset.((unfold s').contains)n
                                    else ...
```

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.

2 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:

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

The sig 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 what they are. The type $e.T$ is a *dependent type*, because it depends on a term whose value is decided at runtime.

We extend the set of expressions as follows

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

Now we can abstract data types in a satisfactory way.

2.1 Examples

A module that implements rationals:

```
RATIONAL ≜ interface{type T;
                     val create : int * int → T
                         add : T * T → T}
```

```
let Rational = module{type=int * int;
                      val create = $\lambda p, q : \mathsf{int}.\langle p, q\rangle$,
                          add = $\lambda r_1, r_2 : T.\langle$left $r_1$ * right $r_2$ + left $r_2$ * right $r_1$, right $r_1$ * right $r_2\}$
in Rational.add (Rational.create⟨1, 2⟩)(Rational.create⟨3, 4⟩)
: ratmod.T
```

To the external user, the function add from the module Rational has the type Rational.T $*$ Rational.$T \to$ Rational.T

Rational in the expressions above is not a type but a term. T is a type. A type whose meaning depends on a term is called a *dependent type*.

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 an existential type instead of a module type.

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

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

3 Strong Existential Types

We have been looking at weak existential types, which don't explain what is going on in the module types just given. We can extend to strong existential types by adding a new dependent type $e.T$ that refers to the type contained in the existentially-typed expression e. Symmetrically, $e.V$ refers to the value contained in e.

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

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

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

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

because σ may depend on a term.

The inference rule for this term is:

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

Observe that to check that $e.T$ 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 to allow type variables to escape:

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

Here we don't need the requirement $\Delta \vdash \sigma_2$ since the hidden type Y can be talked about using $e_1.T$.

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 $p_1 = \mathsf{pack}_{\exists X.X*(X \to \mathsf{bool})}[\mathsf{int}, \langle 2, \lambda n.n = 2\rangle]$ in
  let $p_2 = \mathsf{pack}_{\exists X.X*(X \to \mathsf{bool})}[\mathsf{bool}, \langle \#t, \lambda b.b\rangle]$ 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 $e.T$, in this example, which corresponds to $p_1.T$ and $p_2.T$. After substituting Y by $p_1.T$ and $p_2.T$ respectively and type-checking, we find out f has type $p_2.X \to \mathsf{bool}$ 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 $p_1 = \mathsf{pack}_{\exists X.X*(X \to \mathsf{bool})}[\mathsf{int}, \langle 2, \lambda n.n = 2\rangle]$ in
  let $p_2 = \mathsf{pack}_{\exists X.X*(X \to \mathsf{bool})}[\mathsf{bool}, \langle \#t, \lambda b.b\rangle]$ 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 $p_1.X \to \mathsf{bool}$ 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.