CS 4110

Programming Languages & Logics

Lecture 27

Existential Types

5 November 2014

Announcements

- Tomorrow: *Reading Code as if it Mattered* Yaron Minsky PhD '02
  2pm in Gate 310

  Abstract: This talk describes the design of a new system for managing the process of developing, reviewing and releasing software, called Iron. Iron runs on top of a traditional DVCS, in our case Mercurial, and is responsible for managing a patch-oriented review process on top of that DVCS that precisely tracks what people have read and need to read. Notably, Iron makes it easy to deal with merges, including conflicted merges. In addition, Iron provides a hierarchical relationship between different features that allows handling of multiple release processes within the same codebase, and also manage more complex workflows like handling concurrent review of features where one feature depends on the other.

- HW 7 due tomorrow
- HW 8 out today

Type Abstraction

We've now seen several examples of type abstraction:

- Simple types
- Polymorphism
- Subtyping

These types enforce a discipline over the code untyped code that, in many situations, provides useful abstractions for programmers

Today we'll look at another type system feature, *existential types*, that provides a way to abstract aspects of types themselves

Namespaces

Simple languages, like C and FORTRAN, often have a single global namespace

This causes problems in large programs due to name collisions—i.e., two different programmers (or pieces of code) using the same name for different purposes—are likely

In addition, it often leads to situations where components of a program are tightly coupled, since one component may use a name defined by the other

Modularity

Modular programming addresses these issues

A *module* is a collection of named entities that are related to each other in some way

Modules provide separate namespaces: different modules have different name spaces, and so can freely use names without worrying about name collisions

Some essential module features:

- Choose which names to export
- Choose which names to keep hidden
- Hide implementation details

Existential Types

$$
\begin{aligned}
\tau ::= {} & \mathbf{int} \\
& \mid \tau_1 \rightarrow \tau_2 \\
& \mid \{\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\} \\
& \mid \exists X.\ \tau \\
& \mid X
\end{aligned}
$$

An existential type is written $\exists X.\ \tau$, where type variable X may occur in τ

If a value has type $\exists X.\ \tau$, it means that it is a pair $\{\tau', v\}$ of a type τ' and a value v, such that v has type $\tau\{\tau'/X\}$

Syntax

$$
\begin{aligned}
e ::= {} & x \\
| {} & \lambda x{:}\tau.\, e \\
| {} & e_1\ e_2 \\
| {} & n \\
| {} & e_1 + e_2 \\
| {} & \{\, l_1 = e_1, \ldots, l_n = e_n \,\} \\
| {} & e.l \\
| {} & \text{pack}\ \{\tau_1, e\}\ \text{as}\ \exists X.\, \tau_2 \\
| {} & \text{unpack}\ \{X, x\} = e_1\ \text{in}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
v ::= {} & n \\
| {} & \lambda x{:}\tau.\, e \\
| {} & \{\, l_1 = v_1, \ldots, l_n = v_n \,\} \\
| {} & \text{pack}\ \{\tau_1, v\}\ \text{as}\ \exists X.\, \tau_2
\end{aligned}
$$

Dynamic Semantics

$$\begin{aligned} E ::= {} & \ldots \\ & \mid \text{pack } \{\tau_1, E\} \text{ as } \exists X.\ \tau_2 \\ & \mid \text{unpack } \{X, x\} = E \text{ in } e \end{aligned}$$

$$\frac{}{\text{unpack } \{X, x\} = (\text{pack } \{\tau_1, v\} \text{ as } \exists Y.\ \tau_2) \text{ in } e \rightarrow e\{v/x\}\{\tau_1/X\}}$$

Static Semantics

$$\frac{\Delta, \Gamma \vdash e : \tau_2\{\tau_1/X\} \qquad \Delta \vdash \exists X.\, \tau_2 \text{ ok}}{\Delta, \Gamma \vdash \text{pack } \{\tau_1, e\} \text{ as } \exists X.\, \tau_2 : \exists X.\, \tau_2}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \exists X.\, \tau_1 \qquad \Delta \cup \{X\}, \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \Delta \vdash \tau_2 \text{ ok}}{\Delta, \Gamma \vdash \text{unpack } \{X, x\} = e_1 \text{ in } e_2 : \tau_2}$$

Note that in the typing rule for unpack, the side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable X does *not* appear free in τ_2

Example

```
let $counterADT$ =
    pack {$\mathbf{int}$,
              { new = 0,
                 get = $\lambda i : \mathbf{int}.\, i$,
                 inc = $\lambda i : \mathbf{int}.\, i + 1$ } }
    as
        $\exists\mathbf{Counter}.$
              { new : $\mathbf{Counter}$,
                 get : $\mathbf{Counter} \to \mathbf{int}$,
                 inc : $\mathbf{Counter} \to \mathbf{Counter}$}
in . . .
```

Example, Continued

We can use the existential value *counterADT* as follows.

$$\begin{array}{l} \text{unpack}\ \{C, c\} = \mathit{counterADT}\ \text{in} \\ \text{let}\ y = c.\text{new in} \\ c.\text{get}\ (c.\text{inc}\ (c.\text{inc}\ y)) \end{array}$$

Example, Continued

We can use the existential value *counterADT* as follows.

$$\begin{array}{l}\text{unpack } \{C, c\} = \mathit{counterADT} \text{ in} \\ \text{let } y = c.\text{new in} \\ c.\text{get } (c.\text{inc } (c.\text{inc } y))\end{array}$$

Note that we annotate the pack construct with the existential type

Why do we do this?

Example, Continued

We can use the existential value *counterADT* as follows.

$$\begin{array}{l}\text{unpack } \{C, c\} = \mathit{counterADT} \text{ in} \\ \text{let } y = c.\text{new in} \\ c.\text{get } (c.\text{inc } (c.\text{inc } y))\end{array}$$

Note that we annotate the pack construct with the existential type

Why do we do this?

Without this annotation, we would not know which occurrences of the witness type are intended to be replaced with the type variable, and which are intended to be left as the witness type

Representation Independence

We can define alternate, equivalent implementations of our counter...

$$
\begin{array}{l}
\text{let } \mathit{counterADT} = \\
\quad \text{pack } \{\{x{:}\,\mathbf{int}\}, \\
\qquad\qquad \{\ \text{new} = \{x = 0\}, \\
\qquad\qquad\quad \text{get} = \lambda r{:}\{x{:}\,\mathbf{int}\}.\ r.x, \\
\qquad\qquad\quad \text{inc} = \lambda r{:}\{x{:}\,\mathbf{int}\}.\ r.x + 1\ \}\ \} \\
\quad \text{as} \\
\qquad \exists \mathbf{Counter}. \\
\qquad\qquad \{\ \text{new} : \mathbf{Counter}, \\
\qquad\qquad\quad \text{get} : \mathbf{Counter} \to \mathbf{int}, \\
\qquad\qquad\quad \text{inc} : \mathbf{Counter} \to \mathbf{Counter}\} \\
\text{in} \ldots
\end{array}
$$

Existentials and Type Variables

Note that in the typing rule for unpack, the side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable X does *not* appear free in τ_2

Existentials and Type Variables

Note that in the typing rule for unpack, the side condition $\Delta \vdash \tau_2$ ok ensures that the existentially quantified type variable X does *not* appear free in τ_2

This rules out programs such as,

let $m =$
$\quad$ pack $\{\textbf{int}, \{a = 5, f = \lambda x{:}\textbf{int}.x + 1\}\}$ as $\exists X.\ \{a{:}X, f{:}X \to X\}$
in
unpack $\{X, x\} = m$ in $x.f\ x.a$

where the type of $(f.x\ x.a)$ has X free

Encoding Existentials

It turns out that we can encode existentials using polymorphism!

The idea is to use a Church encoding, where an existential value is a function that takes a type and then calls the continuation

Encoding Existentials

It turns out that we can encode existentials using polymorphism!

The idea is to use a Church encoding, where an existential value is a function that takes a type and then calls the continuation

$$\begin{aligned}
\exists X.\, \tau \quad &\triangleq \quad \forall Y.\, (\forall X.\, \tau \to Y) \to Y \\
\text{pack } \{\tau_1, e\} \text{ as } \exists X.\, \tau_2 \quad &\triangleq \quad \Lambda Y.\, \lambda f : (\forall X. \tau_2 \to Y).\, f\, [\tau_1]\, e \\
\text{unpack } \{X, x\} = e_1 \text{ in } e_2 \quad &\triangleq \quad e_1\, [\tau_2]\, (\Lambda X. \lambda x : \tau_1.\, e_2)
\end{aligned}$$

where e_1 has type $\exists X. \tau_1$ and e_2 has type τ_2

For more details see Pierce, Chapter 24