

# Type Systems for Programming Languages (15-814)

## Lecture Notes, Fall 2006, Week 8

William Lovas (wlovas@cs)

October 31, 2006

### 1 Existential types

Just as universal types  $\forall X. \tau$  model “generics”, existential types  $\exists X. \tau$  model “abstract types”.

$$\begin{aligned} \tau &::= \dots \mid \exists X. \tau \\ e &::= \dots \mid \mathbf{pack} \sigma \mathbf{with} e \mathbf{as} \exists X. \tau \mid \mathbf{open} e \mathbf{as} X \mathbf{with} x:\tau \mathbf{in} e' \end{aligned}$$

As ABTs:

$$\begin{aligned} &\mathbf{pack}[\exists X. \tau](\sigma, e) \\ &\mathbf{open}(e, X. x:\tau. e') \end{aligned}$$

**Note:**  $X$  and  $x$  are bound in  $e'$ :  $X$  is the representation type,  $x$  is the “operations”.  
For example, we might represent the ML signature

```
signature COUNTER =
sig
  type Counter
  val new : Counter
  val inc : Counter -> Counter
  val get : Counter -> nat
end
```

as the existential type

$$COUNTER = \exists Counter. \{ \mathbf{new} : Counter, \mathbf{inc} : Counter \rightarrow Counter, \mathbf{get} : Counter \rightarrow \mathbf{nat} \}$$

#### 1.1 Implementer side

The implementer packages a representation type with some operations over that type, hiding the representation type behind an existential quantifier.

$$\mathbf{pack} repr \mathbf{with} opns \mathbf{as} \exists X. \tau$$

For example,

$$\mathbf{pack} \mathbf{nat} \mathbf{with} \{ \mathbf{new} = 0, \mathbf{get} = \lambda c:\mathbf{nat}. c, \mathbf{inc} = \lambda c:\mathbf{nat}. \mathbf{suc}c \} \mathbf{as} COUNTER,$$

where  $COUNTER$  is the existential type shown above.

**Note:** The implementer “knows” that  $Counter = \mathbf{nat}$ .

## 1.2 Client side

The client unpacks an existential package, binding its representation type and its operations to variables in a new scope.

**open**  $e$  **as**  $X$  **with**  $x:\tau$  **in**  $e'$

For example, if  $c : \text{COUNTER}$ ,

**open**  $c$  **as**  $\text{Counter}$  **with**  $x:\{\text{new} : \text{Counter}, \dots\}$  **in**  $\dots x.\text{new} \dots x.\text{get} \dots x.\text{inc} \dots$ ,

where the body may refer to the representation type as  $\text{Counter}$  and the operations as  $x$ , but must have an *extrinsically meaningful* type—i.e. one that does not refer to  $\text{Counter}$ .

## 1.3 Statics

$$\frac{\Delta \vdash \sigma \text{ type} \quad \Delta; \Gamma \vdash e : [\sigma/X] \tau \quad \Delta, X \text{ type} \vdash \tau \text{ type}}{\Delta; \Gamma \vdash \text{pack } \sigma \text{ with } e \text{ as } \exists X. \tau : \exists X. \tau} (\exists\text{I})$$

$$\frac{\Delta; \Gamma \vdash e : \exists X. \tau \quad (\Delta, X \text{ type}); (\Gamma, x:\tau) \vdash e' : \tau' \quad \Delta \vdash \tau' \text{ type}}{\Delta; \Gamma \vdash \text{open } e \text{ as } X \text{ with } x:\tau \text{ in } e' : \tau'} (\exists\text{E})$$

The introduction rule  $\exists\text{I}$  creates an existential package of type  $\exists X. \tau$ ; the intuition that the implementer “knows” the representation type  $\sigma$  is captured by typechecking the operations  $e$  at type  $\tau$  with the representation substituted in for the type variable  $X$ .

The elimination rule  $\exists\text{E}$  uses an existential package  $e$  in a body  $e'$ ; the intuition that the client does not know the representation type is captured by its being a variable. Furthermore, the notion that the result type  $\tau'$  must be extrinsically meaningful is captured by the premise  $\Delta \vdash \tau' \text{ type}$ , which says that  $\tau'$  must be a type in the outer context, which does not include  $X$ .

## 1.4 Dynamics

$$\frac{e_1 \mapsto e'_1}{\text{open } e_1 \text{ as } X \text{ with } x:\tau \text{ in } e_2 \mapsto \text{open } e'_1 \text{ as } X \text{ with } x:\tau \text{ in } e_2}$$

$$\frac{}{\text{open } (\text{pack } \sigma \text{ with } e \text{ as } \exists X. \tau) \text{ as } X \text{ with } x:\tau \text{ in } e' \mapsto [\sigma/X][e/x]e'}$$

Notice how opening an existential package substitutes the representation type and the operations for the appropriate bound variables inside the body. The abstraction boundary is erased at run-time!

Three observations:

1. Data abstraction is only a *protocol*, imposed on yourself.
2. There is no abstraction at run-time! Data abstraction is a *compile-time* discipline.
3. Client code is polymorphic! As if “ $e' : \forall X. \tau \rightarrow \tau'$ ”.

The last point leads us to see that we may actually define existentials using only universal types.

## 1.5 Definability of existentials

$$\begin{aligned} \exists X. \tau &:= \forall X'. (\forall X. \tau \rightarrow X') \rightarrow X' \\ \text{pack } \sigma \text{ with } e \text{ as } \exists X. \tau &:= \Lambda X'. \lambda h: (\forall X. \tau \rightarrow X'). h [\sigma] e \\ \text{open } e \text{ as } X \text{ with } x:\tau \text{ in } e' &:= e [\tau'] (\Lambda X. \lambda x:\tau. e') \end{aligned}$$

An existential package is simply something which for any result type, given a polymorphic client producing that result type, can produce that result type. Note two interesting facts:

1. As mentioned above, the client code is polymorphic—it cannot depend on the representation type, because it must work for *any* representation type.
2. In the elimination form, the result type  $\tau'$  is *automatically* extrinsically meaningful, since it occurs outside the scope of  $X$ , which is only bound inside the client code.

This encoding follows the usual principles for defining types in System F: a type is defined by what it can do, using something like a “visitor pattern”.