Type Systems

Lecture 6: Existentials, Data Abstraction, and Termination for System F

Neel Krishnaswami
University of Cambridge
Polymorphism and Data Abstraction

• So far, we have used polymorphism to model datatypes and genericity

• Reynolds’s original motivation was to model data abstraction
An ML Module Signature

module type BOOL = sig
  type t
  val yes : t
  val no : t
  val choose : t -> 'a -> 'a -> 'a
end

• We introduce an abstract type \( t \)
• There are two values, \( \text{yes} \) and \( \text{no} \) of type \( t \)
• There is an operation \( \text{choose} \), which takes a \( t \) and two values, and switches between them.
module M1 : BOOL = struct
  type t = unit option
  let yes = Some ()
  let no = None
  let choose v ifyes ifno =
    match v with
    | Some () -> ifyes
    | None -> ifno
end

- Implementation uses option type over unit
- There are two values, one for true and one for false
- choose implemented via pattern matching
Another Implementation

```plaintext
module M2 : BOOL = struct

  type t = int

  let yes = 1
  let no = 0

  let choose b ifyes ifno =
      if b = 1 then ifyes else ifno

end
```

- Implement booleans with integers
- Use 1 for true, 0 for false
- Why is this okay? (Many more integers than booleans, after all)
module M3 : BOOL = struct
  type t =
    {f : 'a. 'a -> 'a -> 'a}.
  let yes =
    {f = fun a b -> a}
  let no =
    {f = fun a b -> b}
  let choose b ifyes ifno =
    b.f ifyes ifno
end

• Implement booleans with Church encoding (plus some Ocaml hacks)
  • Is this really the same type as in the previous lecture?
A Common Pattern

- We have a signature — **BOOL** — with an abstract type in it
- We choose a concrete implementation of that abstract type
- We implement the other operations (**yes, no, choose**) of the interface in terms of that concrete representation
- Client code cannot identify the representation type because it sees an abstract type variable `t` rather than the representation
Abstract Data Types in System F

Types  \[ A ::= \ldots \mid \exists \alpha. A \]

Terms  \[ e ::= \ldots \mid \text{pack}_{\alpha. B}(A, e) \mid \text{let \ pack}(\alpha, x) = e \text{ in } e' \]

Values  \[ v ::= \text{pack}_{\alpha. B}(A, v) \]

\[
\begin{align*}
\Theta, \alpha \vdash B & \text{ type } & \Theta \vdash A & \text{ type } & \Theta; \Gamma \vdash e : [A/\alpha]B \\
\Theta; \Gamma \vdash \text{pack}_{\alpha. B}(A, e) & : \exists \alpha. B \\
\Theta; \Gamma \vdash e : \exists \alpha. A & & \Theta, \alpha; \Gamma, x : A \vdash e' : C & & \Theta \vdash C & \text{ type } \\
\Theta; \Gamma \vdash \text{let \ pack}(\alpha, x) = e \text{ in } e' : C & & \exists E
\end{align*}
\]
Operational Semantics for Abstract Types

\[
\begin{align*}
\text{let pack}(\alpha, x) = e & \text{ in } t \\
& \sim \text{ let pack}(\alpha, x) = e' \text{ in } t \\
\text{let pack}(\alpha, x) = \text{pack}_{\alpha.\mathcal{B}}(A, v) \text{ in } e & \sim [A/\alpha, v/x]e
\end{align*}
\]
We have a signature with an abstract type in it

We choose a concrete implementation of that abstract type

We implement the operations of the interface in terms of the concrete representation

Client code sees an abstract type variable \( \alpha \) rather than the representation
Abstract Types Have Existential Type

- No accident we write $\exists \alpha. B$ for abstract types!
- This is exactly the same thing as existential quantification in second-order logic
- Discovered by Mitchell and Plotkin in 1988 – *Abstract Types Have Existential Type*
- But Reynolds was thinking about data abstraction in 1976...?
A Church Encoding for Existential Types

\[
\frac{\Theta, \alpha \vdash B \text{ type} \quad \Theta \vdash A \text{ type} \quad \Theta; \Gamma \vdash e : [A/\alpha]B}{\Theta; \Gamma \vdash \text{pack}_{\alpha. B}(A, e) : \exists \alpha. B}
\]

\[
\frac{\Theta; \Gamma \vdash e : \exists \alpha. B \quad \Theta, \alpha; \Gamma, x : B \vdash e' : C \quad \Theta \vdash C \text{ type}}{\Theta; \Gamma \vdash \text{let pack}(\alpha, x) = e \text{ in } e' : C}
\]

<table>
<thead>
<tr>
<th>Original</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists \alpha. B )</td>
<td>( \forall \beta. (\forall \alpha. B \rightarrow \beta) \rightarrow \beta )</td>
</tr>
<tr>
<td>pack(_{\alpha. B})(A, e)</td>
<td>( \land \beta. \lambda k : \forall \alpha. B \rightarrow \beta. k A e )</td>
</tr>
<tr>
<td>let pack((\alpha, x)) = e in e' : C</td>
<td>e C ((\land \alpha. \lambda x : B. e'))</td>
</tr>
</tbody>
</table>
let pack(α, x) = pack_{α.B}(A, e) in e' : C
= pack_{α.B}(A, e) C (\Lambdaα. λx : B. e')
= (\Lambdaβ. λk : \forallα. B \to β. k A e) C (\Lambdaα. λx : B. e')
= (λk : \forallα. B \to C. k A e) (\Lambdaα. λx : B. e')
= (\Lambdaα. λx : B. e') A e 
= (λx : [A/α]B. [A/α]e') e 
= [e/x][A/α]e'
System F, The Girard-Reynolds Polymorphic Lambda Calculus

Types

\[ A ::= \alpha \mid A \rightarrow B \mid \forall \alpha. A \]

Terms

\[ e ::= x \mid \lambda x : A. e \mid ee \mid \Lambda \alpha. e \mid eA \]

Values

\[ v ::= \lambda x : A. e \mid \Lambda \alpha. e \]

\[ e_0 \sim e'_0 \quad \text{CONGFUN} \]

\[ e_0 e_1 \sim e'_0 e_1 \]

\[ e_1 \sim e'_1 \quad \text{CONGFUNARG} \]

\[ v_0 e_1 \sim v_0 e'_1 \]

\[ (\lambda x : A. e) v \sim [v/x]e \quad \text{FUNEVAL} \]

\[ e \sim e' \quad \text{CONGFORALL} \]

\[ eA \sim e'A \]

\[ e A \sim e'A \quad \text{CONGFORALL} \]

\[ (\Lambda \alpha. e) A \sim [A/\alpha]e \quad \text{FORALLEVAL} \]
So far:

1. We have seen System F and its basic properties
2. Sketched a proof of type safety
3. Saw that a variety of datatypes were encodable in it
4. We saw that even data abstraction was representable in it
5. We asserted, but did not prove, termination
• We proved termination for the STLC by defining a logical relation
  • This was a family of relations
  • Relations defined by recursion on the structure of the type
  • Enforced a “hereditary termination” property

• Can we define a logical relation for System F?
  • How do we handle free type variables? (i.e., what’s the interpretation of $\alpha$?)
  • How do we handle quantifiers? (i.e., what’s the interpretation of $\forall \alpha A$?)
A *semantic type* is a set of closed terms $X$ such that:

- (Halting) If $e \in X$, then $e$ halts (i.e. $e \leadsto^* v$ for some $v$).
- (Closure) If $e \leadsto e'$, then $e' \in X$ iff $e \in X$.

Idea:

- Build generic properties of the logical relation into the definition of a type.
- Use this to interpret variables!
We can interpret type well-formedness derivations.

Given a type variable context $\Theta$, we define an interpretation $\theta$ as a map from $\text{dom}(\Theta)$ to semantic types.
Interpretation of Types

\[ [\Theta \vdash \alpha \text{ type}] \theta \quad = \quad \theta(\alpha) \]

\[ [\Theta \vdash A \rightarrow B \text{ type}] \theta \quad = \quad \left\{ e \mid \begin{array}{l} e \text{ halts} \land \\ \forall e' \in [\Theta \vdash A \text{ type}] \theta. \\ (e \ e') \in [\Theta \vdash B \text{ type}] \theta \end{array} \right\} \]

\[ [\Theta \vdash \forall \alpha. B \text{ type}] \theta \quad = \quad \left\{ e \mid \begin{array}{l} e \text{ halts} \land \\ \forall A, X \in \text{SemType}. \\ (e A) \in [\Theta, \alpha \vdash B \text{ type}] (\theta, X/\alpha) \end{array} \right\} \]

Note the lack of a link between $A$ and $X$ in the $\forall \alpha. B$ case.
Properties of the Interpretation

- **Closure**: If $\theta$ is an interpretation for $\Theta$, then $[[\Theta \vdash A \text{ type}]] \theta$ is a semantic type.
- **Exchange**: $[[\Theta, \alpha, \beta, \Theta' \vdash A \text{ type}]] = [[\Theta, \beta, \alpha, \Theta' \vdash A \text{ type}]]$
- **Weakening**: If $\Theta \vdash A \text{ type}$, then $[[\Theta, \alpha \vdash A \text{ type}]] (\theta, X/\alpha) = [[\Theta \vdash A \text{ type}]] \theta$.
- **Substitution**: If $\Theta \vdash A \text{ type}$ and $\Theta, \alpha \vdash B \text{ type}$ then $[[\Theta \vdash [A/\alpha]B \text{ type}]] \theta = [[\Theta, \alpha \vdash B \text{ type}]] (\theta, [[\Theta \vdash A \text{ type}]] \theta)$

Each property is proved by induction on a type well-formedness derivation.
Closure: (one half of the) ∀ Case

Closure: If $\theta$ interprets $\Theta$, then $[\Theta \vdash \forall \alpha. A \text{ type}] \theta$ is a type.

Suffices to show: if $e \sim e'$, then $e \in [\Theta \vdash \forall \alpha. A \text{ type}] \theta$ iff $e' \in [\Theta \vdash \forall \alpha. A \text{ type}] \theta$.

0  $e \sim e'$  Assumption
1  $e' \in [\Theta \vdash \forall \alpha. A \text{ type}] \theta$  Assumption
2  $\forall (C, X). e' C \in [\Theta, \alpha \vdash A \text{ type}] (\theta, X/\alpha)$  Def.
3  Assume $(C, X)$
4  $e' C \in [\Theta, \alpha \vdash A \text{ type}] (\theta, X/\alpha)$  By 2
5  $e C \sim e' C$  CONGFORALL on 0
6  $e C \in [\Theta, \alpha \vdash A \text{ type}] (\theta, X/\alpha)$  Induction on 4,5
7  $\forall (C, X). e C \in [\Theta, \alpha \vdash A \text{ type}] (\theta, X/\alpha)$
8  $e \in [\Theta \vdash \forall \alpha. A \text{ type}] \theta$  From 7
Substitution: (one half of) the $\forall$ case

$\Gamma, \alpha \vdash \forall \beta. B$ type $\quad (\theta, \Gamma \vdash A$ type $\quad \theta) = \Gamma \vdash [A/\alpha](\forall \beta. B)$ type $\quad \theta$

1. We assume $e \in \Gamma, \alpha \vdash \forall \beta. B$ type $\quad (\theta, \Gamma \vdash A$ type $\quad \theta)$
2. We want to show: $e \in \Gamma \vdash [A/\alpha](\forall \beta. B)$ type $\quad \theta$.
3. So from 1:
   $\forall (C, X). \ e \ C \in \Gamma, \alpha, \beta \vdash B$ type $\quad (\theta, \Gamma \vdash A$ type $\quad \theta, X/\beta)$.
4. For 2, it suffices to show:
   $\forall (C, X). \ e \ C \in \Gamma, \beta \vdash [A/\alpha](B)$ type $\quad (\theta, X/\beta)$.
   - Assume $(C, X)$
   - So $e \ C \in \Gamma, \alpha, \beta \vdash B$ type $\quad (\theta, \Gamma \vdash A$ type $\quad \theta, X/\beta)$
   - Exchange: $e \ C \in \Gamma, \beta, \alpha \vdash B$ type $\quad (\theta, X/\beta, \Gamma \vdash A$ type $\quad \theta)$
   - Weaken:
     $e \ C \in \Gamma, \beta, \alpha \vdash B$ type $\quad (\theta, X/\beta, \Gamma \vdash A$ type $\quad (\theta, X/\beta))$
   - Induction: $e \ C \in \Gamma, \beta \vdash [A/\alpha]B$ type $\quad (\theta, X/\beta)$
If we have that

\[
\Theta, \Gamma \vdash \alpha_1, \ldots, \alpha_k; x_1 : A_1, \ldots, x_n : A_n \vdash e : B
\]

\[
\Theta \vdash \Gamma \text{ ctx}
\]

\[
\theta \text{ interprets } \Theta
\]

\[
\text{For each } x_i : A_i \in \Gamma, \text{ we have } e_i \in \llbracket \Theta \vdash A_i \text{ type} \rrbracket \theta
\]

Then it follows that:

\[
\llbracket C_1/\alpha_1, \ldots, C_k/\alpha_k \rrbracket [e_1/x_1, \ldots, e_n/x_n] e \in \llbracket \Theta \vdash B \text{ type} \rrbracket \theta
\]
1. Prove the other direction of the closure property for the \( \Theta \vdash \forall \alpha. A \) type case.

2. Prove the other direction of the substitution property for the \( \Theta \vdash \forall \alpha. A \) type case.

3. Prove the fundamental lemma for the forall-introduction case \( \Theta; \Gamma \vdash \forall \alpha. e : \forall \alpha. A \).