Type Systems

Lecture 4: Datatypes and Polymorphism

Neel Krishnaswami
University of Cambridge
• One of the essential features of programming languages is data
• So far, we have sums and product types
• This is enough to represent basic datatypes
### Booleans

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 + 1</td>
</tr>
<tr>
<td>true</td>
<td>L ⟨⟩</td>
</tr>
<tr>
<td>false</td>
<td>R ⟨⟩</td>
</tr>
<tr>
<td>if e then e' else e''</td>
<td>case(e, L _ → e', R _ → e'')</td>
</tr>
</tbody>
</table>

\[ \Gamma \vdash \text{true : bool} \quad \Gamma \vdash \text{false : bool} \]

\[ \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : X \quad \Gamma \vdash e'' : X \]

\[ \Gamma \vdash \text{if e then e' else e'' : X} \]
### Characters

<table>
<thead>
<tr>
<th>Builtin</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>bool$^7$ (for ASCII!)</td>
</tr>
<tr>
<td>'A'</td>
<td>(true, false, false, false, false, false, false, true)</td>
</tr>
<tr>
<td>'B'</td>
<td>(true, false, false, false, false, false, true, false)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- This is not a wieldy encoding!
- But it works, more or less
- Example: define equality on characters
Limitations

The STLC gives us:

- Representations of data
- The ability to do conditional branches on data
- The ability to do functional abstraction on operations
- **MISSING:** the ability to loop
Unbounded Recursion = Inconsistency

\[ \Gamma, f : X \rightarrow Y, x : X \vdash e : Y \]
\[ \Gamma \vdash \text{fun}_{X \rightarrow Y} fx. e : X \rightarrow Y \]
\[ \text{FIX} \]

\[ e' \leadsto e'' \]
\[ (\text{fun}_{X \rightarrow Y} fx. e) e' \leadsto (\text{fun}_{X \rightarrow Y} fx. e) e'' \]
\[ (\text{fun}_{X \rightarrow Y} fx. e) v \leadsto [\text{fun}_{X \rightarrow Y} fx. e/f, v/x]e \]

- Modulo type inference, this is basically the typing rule Ocaml uses
- It permits defining recursive functions very naturally
The Typing of a Perfectly Fine Factorial Function

\[
\Delta \vdash fact : \text{int} \rightarrow \text{int} \quad \Delta \vdash n - 1 : \text{int} \\
\text{...}
\]

\[
\Delta \vdash fact(n - 1) : \text{int} \\
\text{...}
\]

\[
\Delta \vdash n \times fact(n - 1) : \text{int} \\
\]

\[
\Delta \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times fact(n - 1) : \text{int} \\
\]

\[
\Gamma, fact : \text{int} \rightarrow \text{int}, n : \text{int} \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times fact(n - 1) : \text{int} \\
\]

\[
\Gamma \vdash \text{fun}_{\text{int} \rightarrow \text{int}} fact \ n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times fact(n - 1) : \text{int} \rightarrow \text{int} \\
\]
A Bad Use of Recursion

\[
\begin{align*}
  f : 1 \rightarrow 0, x : 1 \vdash f : 1 \rightarrow 0 \\
  f : 1 \rightarrow 0, x : 1 \vdash x : 1 \\
  f : 1 \rightarrow 0, x : 1 \vdash fx : 0 \\
  \vdash \text{fun} \, 1 \rightarrow 0 \, fx. \, fx : 1 \rightarrow 0
\end{align*}
\]

\[
(\text{fun} \, 1 \rightarrow 0 \, fx. \, fx) \langle \rangle \ \sim \ \left[\text{fun} \, 1 \rightarrow 0 \, fx. \, fx/f, \langle \rangle /x\right](fx)
\]

\[
\equiv \ (\text{fun} \, 1 \rightarrow 0 \, fx. \, fx) \langle \rangle
\]

\[
\sim \ \left[\text{fun} \, 1 \rightarrow 0 \, fx. \, fx/f, \langle \rangle /x\right](fx)
\]

\[
\equiv \ (\text{fun} \, 1 \rightarrow 0 \, fx. \, fx) \langle \rangle
\]

\[
\ldots
\]
• Iteration looks like a bounded for-loop
• It is surprisingly expressive:

\[ n + m \triangleq \text{iter}(n, z \rightarrow m, s(x) \rightarrow s(x)) \]
\[ n \times m \triangleq \text{iter}(n, z \rightarrow z, s(x) \rightarrow m + x) \]
\[ \text{pow}(n, m) \triangleq \text{iter}(m, z \rightarrow s(z), s(x) \rightarrow n \times x) \]

• These definitions are \textit{primitive recursive}
• Our language is more expressive!
The Ackermann-Péter Function

\[ A(0, n) = n + 1 \]
\[ A(m + 1, 0) = A(m, 1) \]
\[ A(m + 1, n + 1) = A(m, A(m + 1, n)) \]

- One of the simplest fast-growing functions
- It’s not “primitive recursive” (we won’t prove this)
- However, it does terminate
  - Either \( m \) decreases (and \( n \) can change arbitrarily), or
  - \( m \) stays the same and \( n \) decreases
  - Lexicographic argument
repeat : \( (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \)
repeat \( \triangleq \lambda f. \lambda n. \text{iter}(n, z \to f, s(x) \to f \circ x) \)

ack : \( \mathbb{N} \to \mathbb{N} \to \mathbb{N} \)
ack \( \triangleq \lambda m. \lambda n. \text{iter}(m, z \to (\lambda x. s(x)), s(r) \to \text{repeat } r) \, n \)

- Proposition: \( A(n, m) \triangleq \text{ack } n \, m \)
- Note the critical use of iteration at “higher type”
- Despite totality, the calculus is extremely powerful
- Functional programmers call things like iter recursion schemes
\[\Gamma \vdash \text{ListNil} \quad \Gamma \vdash e : X \quad \Gamma \vdash e' : \text{list}X \quad \Gamma \vdash \text{ListCons} \\]

\[\Gamma \vdash e_0 : \text{list}X \quad \Gamma \vdash e_1 : Z \quad \Gamma, x : X, r : Z \vdash e_2 : Z \quad \Gamma \vdash \text{ListFold} \\]

\[\Gamma \vdash \text{fold}(e_0, [] \to e_1, x :: r \to e_2) : Z\]
\[
\begin{align*}
  e_0 & \leadsto e'_0 \\
  e_0 :: e_1 & \leadsto e'_0 :: e_1 \\
  e_1 & \leadsto e'_1 \\
  v_0 :: e_1 & \leadsto v_0 :: e'_1
\end{align*}
\]

\[
\begin{align*}
  e_0 & \leadsto e'_0 \\
  \text{fold}(e_0, [] \rightarrow e_1, x :: r \rightarrow e_2) & \leadsto \text{fold}(e'_0, [] \rightarrow e_1, x :: r \rightarrow e_2) \\
  \text{fold}([], []) \rightarrow e_1, x :: r \rightarrow e_2) & \leadsto e_1 \\
  R & \triangleq \text{fold}(v', [], e_1, x :: r \rightarrow e_2) \\
  \text{fold}(v :: v', [], e_1, x :: r \rightarrow e_2) & \leadsto [v/x, R/r]e_2
\end{align*}
\]
Some Functions on Lists

\[
\text{length} : \text{list } X \rightarrow \mathbb{N} \\
\text{length} \triangleq \lambda x s. \text{fold}(x s, \emptyset \rightarrow z, x :: r \rightarrow s(r))
\]

\[
\text{append} : \text{list } X \rightarrow \text{list } X \rightarrow \text{list } X \\
\text{append} \triangleq \lambda x. \lambda y s. \text{fold}(x s, \emptyset \rightarrow y s, x :: r \rightarrow x :: r)
\]

\[
\text{map} : (X \rightarrow Y) \rightarrow \text{list } X \rightarrow \text{list } Y \\
\text{map} \triangleq \lambda f. \lambda x s. \text{fold}(x s, \emptyset \rightarrow \emptyset, x :: r \rightarrow (f x) :: r)
\]
• The Curry-Howard Correspondence tells us to think of types as propositions.
• But what logical propositions do \( \mathbb{N} \) or list\( X \), correspond to?
• The following biconditionals hold:
  • \( 1 \iff \mathbb{N} \)
  • \( 1 \iff \text{list}\( X \) \)
  • \( \mathbb{N} \iff \text{list}\( X \) \)
• So \( \mathbb{N} \) is “equivalent to” truth?
A Practical Perversity

map : (X → Y) → list X → list Y
map ≜ λf. λxs. fold(xs, [], [] → [], x :: r → (f x) :: r)

• This definition is *schematic* – it tells us how to define map for each pair of types X and Y
• However, when writing programs in the STLC+lists, we must re-define map for each function type we want to apply it at
• This is annoying, since the definition will be *identical* save for the types
The Polymorphic Lambda Calculus

Types \[ A ::= \alpha \mid A \to B \mid \forall \alpha. A \]

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

- We want to support \textit{type polymorphism}
  - append : \( \forall \alpha. \text{list } \alpha \to \text{list } \alpha \to \text{list } \alpha \)
  - map : \( \forall \alpha. \forall \beta. (\alpha \to \beta) \to \text{list } \alpha \to \text{list } \beta \)

- To do this, we introduce \textit{type variables} and \textit{type polymorphism}

- Invented (twice!) in the early 1970s
  - By the French logician Jean-Yves Girard (1972)
  - By the American computer scientist John C. Reynolds (1974)
Well-formedness of Types

Type Contexts

\[ \Theta ::= \cdot \mid \Theta, \alpha \]

- \( \alpha \in \Theta \) \implies \( \Theta \vdash \alpha \text{ type} \)
- \( \Theta \vdash A \text{ type} \) \implies \( \Theta \vdash B \text{ type} \)

\( \Theta, \alpha \vdash A \text{ type} \) \implies \( \Theta \vdash A \rightarrow B \text{ type} \)

• Judgement \( \Theta \vdash A \text{ type} \) checks if a type is well-formed
• Because types can have free variables, we need to check if a type is well-scoped
Well-formedness of Term Contexts

Term Variable Contexts \( \Gamma ::= \cdot \mid \Gamma, x : A \)

\( \Theta \vdash \cdot \) \hspace{2cm} \( \Theta \vdash \Gamma \) \hspace{2cm} \( \Theta \vdash A \) \hspace{2cm} \( \Theta \vdash \Gamma, x : A \) 

- Judgement \( \Theta \vdash \Gamma \) type checks if a term context is well-formed
- We need this because contexts associate variables with types, and types now have a well-formedness condition
Typing for System F

\[
\frac{x : A \in \Gamma}{\Theta; \Gamma \vdash x : A}
\]

\[
\frac{\Theta \vdash A \text{ type} \quad \Theta; \Gamma, x : A \vdash e : B}{\Theta; \Gamma \vdash \lambda x : A. e : A \rightarrow B}
\]

\[
\frac{\Theta; \Gamma \vdash e : A \rightarrow B \quad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e \, e' : B}
\]

\[
\frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \lambda \alpha. e : \forall \alpha. B}
\]

\[
\frac{\Theta; \Gamma \vdash e : \forall \alpha. B}{\Theta; \Gamma \vdash e \, A : [A/\alpha]B}
\]

• Note the presence of substitution in the typing rules!
Ultimately, we want to prove type safety for System F.
However, the introduction of type variables means that a fair amount of additional administrative overhead is introduced.
This may look intimidating on first glance, BUT really it’s all just about keeping track of the free variables in types.
As a result, none of these lemmas are hard – just a little tedious.
1. (Type Weakening) If $\Theta, \Theta' \vdash A$ type then $\Theta, \beta, \Theta' \vdash A$ type.

2. (Type Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash A$ type then $\Theta, \gamma, \beta, \Theta' \vdash A$ type.

3. (Type Substitution) If $\Theta \vdash A$ type and $\Theta, \alpha \vdash B$ type then $\Theta \vdash [A/\alpha]B$ type.

• These follow the pattern in lecture 1, except with fewer cases.
• Needed to handle the type application rule.
1. (Context Weakening) If $\Theta, \Theta' \vdash \Gamma$ ctx then $\Theta, \alpha, \Theta' \vdash \Gamma$ ctx

2. (Context Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma$ ctx then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma$ ctx

3. (Context Substitution) If $\Theta \vdash A$ type and $\Theta, \alpha \vdash \Gamma$ type then $\Theta \vdash [A/\alpha]\Gamma$ type

• This just lifts the type-level structural properties to contexts
Regularity: If $\Theta \vdash \Gamma \text{ ctx}$ and $\Theta; \Gamma \vdash e : A$ then $\Theta \vdash A$ type

Proof: By induction on the derivation of $\Theta; \Gamma \vdash e : A$

• This just says if typechecking succeeds, then it found a well-formed type
• (Type Weakening of Terms) If $\Theta, \Theta' \vdash \Gamma$ ctx and $\Theta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \alpha, \Theta'; \Gamma \vdash e : A$.

• (Type Exchange of Terms) If $\Theta, \alpha, \beta, \Theta' \vdash \Gamma$ ctx and $\Theta, \alpha, \beta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \beta, \alpha, \Theta'; \Gamma \vdash e : A$.

• (Type Substitution of Terms) If $\Theta, \alpha \vdash \Gamma$ ctx and $\Theta \vdash A$ type and $\Theta, \alpha; \Gamma \vdash e : B$ then $\Theta; [A/\alpha] \Gamma \vdash [A/\alpha]e : [A/\alpha]B$.  

Structural Properties and Substitution of Types into Terms
Structural Properties and Substitution for Term Variables

• (Weakening of Terms) If $\Theta \vdash \Gamma, \Gamma'$ ctx and $\Theta \vdash B$ type and $\Theta; \Gamma, \Gamma' \vdash e : A$ then $\Theta; \Gamma, y : B, \Gamma' \vdash e : A$

• (Exchange of Terms) If $\Theta \vdash \Gamma, y : B, z : C, \Gamma'$ ctx and $\Theta; \Gamma, y : B, z : C, \Gamma' \vdash e : A$, then $\Theta; \Gamma, z : C, y : B, \Gamma' \vdash e : A$

• (Substitution of Terms) If $\Theta \vdash \Gamma, x : A$ ctx and $\Theta; \Gamma \vdash e : A$ and $\Theta; \Gamma, x : A \vdash e' : B$ then $\Theta; \Gamma \vdash [e/x]e' : B$.

• There are two sets of substitution theorems, since there are two contexts

• We also need to assume well-formedness conditions

• But the proofs are all otherwise similar
Conclusion

• We have seen how data works in the pure lambda calculus
• We have started to make it more useful with polymorphism
• But where did the data go in System F? (Next lecture!)