

Type Systems

Lecture 4: Datatypes and Polymorphism

Neel Krishnaswami
University of Cambridge

Data Types in the Simply Typed Lambda Calculus

- 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

Builtin	Encoding
bool	$1 + 1$
true	$L \langle \rangle$
false	$R \langle \rangle$
if e then e' else e''	$\text{case}(e, L_ \rightarrow e', R_ \rightarrow e'')$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}$$
$$\frac{}{\Gamma \vdash \text{false} : \text{bool}}$$
$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : X \quad \Gamma \vdash e'' : X}{\Gamma \vdash \text{if } e \text{ then } e' \text{ else } e'' : X}$$

Characters

Builtin	Encoding
char	bool ⁷ (for ASCII!)
'A'	(true, false, false, false, false, false, true)
'B'	(true, false, false, false, false, true, false)
...	...

- 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

$$\frac{\Gamma, f : X \rightarrow Y, x : X \vdash e : Y}{\Gamma \vdash \text{fun}_{X \rightarrow Y} f x. e : X \rightarrow Y} \text{Fix}$$

$$\frac{e' \rightsquigarrow e''}{(\text{fun}_{X \rightarrow Y} f x. e) e' \rightsquigarrow (\text{fun}_{X \rightarrow Y} f x. e) e''}$$

$$\frac{}{(\text{fun}_{X \rightarrow Y} f x. e) v \rightsquigarrow [\text{fun}_{X \rightarrow Y} f x. e / f, v / x] e}$$

- 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

$$\begin{array}{c}
 \vdots \\
 \hline
 \Delta \vdash \text{fact} : \text{int} \rightarrow \text{int} \quad \Delta \vdash n - 1 : \text{int} \\
 \hline
 \dots \quad \Delta \vdash \text{fact}(n - 1) : \text{int} \\
 \hline
 \dots \quad \Delta \vdash n \times \text{fact}(n - 1) : \text{int} \\
 \hline
 \hline
 \overbrace{\Gamma, \text{fact} : \text{int} \rightarrow \text{int}, n : \text{int}}^{\Delta} \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n - 1) : \text{int} \\
 \hline
 \Gamma \vdash \text{fun}_{\text{int} \rightarrow \text{int}} \text{fact } n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n - 1) : \text{int}
 \end{array}$$

A Bad Use of Recursion

$$\frac{\frac{}{f: 1 \rightarrow 0, x: 1 \vdash f: 1 \rightarrow 0} \quad \frac{}{f: 1 \rightarrow 0, x: 1 \vdash x: 1}}{f: 1 \rightarrow 0, x: 1 \vdash fx: 0}}{\cdot \vdash \text{fun}_{1 \rightarrow 0} fx. fx: 1 \rightarrow 0}$$

$$\begin{aligned} (\text{fun}_{1 \rightarrow 0} fx. fx) \langle \rangle &\rightsquigarrow [\text{fun}_{1 \rightarrow 0} fx. fx / f, \langle \rangle / x](fx) \\ &\equiv (\text{fun}_{1 \rightarrow 0} fx. fx) \langle \rangle \\ &\rightsquigarrow [\text{fun}_{1 \rightarrow 0} fx. fx / f, \langle \rangle / x](fx) \\ &\equiv (\text{fun}_{1 \rightarrow 0} fx. fx) \langle \rangle \\ &\dots \end{aligned}$$

Numbers, More Safely

$$\frac{}{\Gamma \vdash z : \mathbb{N}} \text{NI}_z \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash s(e) : \mathbb{N}} \text{NI}_s$$

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \quad \Gamma \vdash e_1 : X \quad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \text{iter}(e_0, z \rightarrow e_1, s(x) \rightarrow e_2) : X} \text{NE}$$

$$\frac{e_0 \rightsquigarrow e'_0}{\text{iter}(e_0, z \rightarrow e_1, s(x) \rightarrow e_2) \rightsquigarrow \text{iter}(e'_0, z \rightarrow e_1, s(x) \rightarrow e_2)}$$

$$\frac{}{\text{iter}(z, z \rightarrow e_1, s(x) \rightarrow e_2) \rightsquigarrow e_1}$$

$$\frac{}{\text{iter}(s(v), z \rightarrow e_1, s(x) \rightarrow e_2) \rightsquigarrow [\text{iter}(v, z \rightarrow e_1, s(x) \rightarrow e_2)/x]e_2}$$

Expressiveness of Gödel's T

- Iteration looks like a bounded for-loop
- It is surprisingly expressive:

$$n + m \quad \triangleq \quad \text{iter}(n, z \rightarrow m, s(x) \rightarrow s(x))$$

$$n \times m \quad \triangleq \quad \text{iter}(n, z \rightarrow z, s(x) \rightarrow m + x)$$

$$\text{pow}(n, m) \quad \triangleq \quad \text{iter}(m, z \rightarrow s(z), s(x) \rightarrow n \times x)$$

- These definitions are *primitive recursive*
- Our language is more expressive!

The Ackermann-Péter Function

$$\begin{aligned}A(0, n) &= n + 1 \\A(m + 1, 0) &= A(m, 1) \\A(m + 1, n + 1) &= A(m, A(m + 1, n))\end{aligned}$$

- 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

The Ackermann-Péter Function in Gödel's T

repeat : $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

repeat $\triangleq \lambda f. \lambda n. \text{iter}(n, z \rightarrow f, s(x) \rightarrow f \circ x)$

ack : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

ack $\triangleq \lambda m. \lambda n. \text{iter}(m, z \rightarrow (\lambda x. s(x)), s(r) \rightarrow \text{repeat } r) n$

- 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*

Data Structures: Lists

$$\begin{array}{c} \frac{}{\Gamma \vdash [] : \text{list } X} \text{LISTNIL} \qquad \frac{\Gamma \vdash e : X \quad \Gamma \vdash e' : \text{list } X}{\Gamma \vdash e :: e' : \text{list } X} \text{LISTCONS} \\[2ex] \frac{\Gamma \vdash e_0 : \text{list } X \quad \Gamma \vdash e_1 : Z \quad \Gamma, x : X, r : Z \vdash e_2 : Z}{\Gamma \vdash \text{fold}(e_0, [] \rightarrow e_1, x :: r \rightarrow e_2) : Z} \text{LISTFOLD} \end{array}$$

Data Structures: Lists

$$\frac{e_0 \rightsquigarrow e'_0}{e_0 :: e_1 \rightsquigarrow e'_0 :: e_1}$$

$$\frac{e_1 \rightsquigarrow e'_1}{v_0 :: e_1 \rightsquigarrow v_0 :: e'_1}$$

$$\frac{e_0 \rightsquigarrow e'_0}{\text{fold}(e_0, [] \rightarrow e_1, x :: r \rightarrow e_2) \rightsquigarrow \text{fold}(e'_0, [] \rightarrow e_1, x :: r \rightarrow e_2)}$$

$$\frac{}{\text{fold}([], [] \rightarrow e_1, x :: r \rightarrow e_2) \rightsquigarrow e_1}$$

$$\frac{R \triangleq \text{fold}(v', [] \rightarrow e_1, x :: r \rightarrow e_2)}{\text{fold}(v :: v', [] \rightarrow e_1, x :: r \rightarrow e_2) \rightsquigarrow [v/x, R/r]e_2}$$

Some Functions on Lists

length : list $X \rightarrow \mathbb{N}$

length $\triangleq \lambda xs. \text{fold}(xs, [] \rightarrow z, x :: r \rightarrow s(r))$

append : list $X \rightarrow \text{list } X \rightarrow \text{list } X$

append $\triangleq \lambda x. \lambda ys. \text{fold}(xs, [] \rightarrow ys, x :: r \rightarrow x :: r)$

map : $(X \rightarrow Y) \rightarrow \text{list } X \rightarrow \text{list } Y$

map $\triangleq \lambda f. \lambda xs. \text{fold}(xs, [] \rightarrow [], x :: r \rightarrow (fx) :: r)$

A Logical Perversity

- 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

$$\begin{aligned}\text{map} &: (X \rightarrow Y) \rightarrow \text{list } X \rightarrow \text{list } Y \\ \text{map} &\triangleq \lambda f. \lambda xs. \text{fold}(xs, [] \rightarrow [], x :: r \rightarrow (fx) :: r)\end{aligned}$$

- 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 $A ::= \alpha \mid A \rightarrow B \mid \forall \alpha. A$

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

- We want to support *type polymorphism*
 - $\text{append} : \forall \alpha. \text{list } \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$
 - $\text{map} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{list } \alpha \rightarrow \text{list } \beta$
- To do this, we introduce *type variables* and *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 $\Theta ::= \cdot \mid \Theta, \alpha$

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}}$$

$$\frac{\Theta \vdash A \text{ type} \quad \Theta \vdash B \text{ type}}{\Theta \vdash A \rightarrow B \text{ type}}$$

$$\frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash A \rightarrow B \text{ type}}$$

- 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 $\Gamma ::= \cdot \mid \Gamma, x : A$

$$\frac{}{\Theta \vdash \cdot \text{ ctx}} \qquad \frac{\Theta \vdash \Gamma \text{ ctx} \quad \Theta \vdash A \text{ type}}{\Theta \vdash \Gamma, x : A \text{ type}}$$

- Judgement $\Theta \vdash \Gamma \text{ type}$ 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

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

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

$$\frac{\Theta; \Gamma \vdash e : A \rightarrow B \quad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e e' : B}$$

$$\frac{\Theta, \alpha; \Gamma \vdash e : B}{\Theta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha. B}$$

$$\frac{\Theta; \Gamma \vdash e : \forall \alpha. B \quad \Theta \vdash A \text{ type}}{\Theta; \Gamma \vdash e A : \boxed{[A/\alpha]B}}$$

- Note the presence of substitution in the typing rules!

The Bookkeeping

- 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

Structural Properties and Substitution for Types

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

Structural Properties and Substitutions for Contexts

1. (Context Weakening) If $\Theta, \Theta' \vdash \Gamma \text{ ctx}$ then $\Theta, \alpha, \Theta' \vdash \Gamma \text{ ctx}$
 2. (Context Exchange) If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma \text{ ctx}$ then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma \text{ ctx}$
 3. (Context Substitution) If $\Theta \vdash A \text{ type}$ and $\Theta, \alpha \vdash \Gamma \text{ type}$ then $\Theta \vdash [A/\alpha]\Gamma \text{ type}$
- This just lifts the type-level structural properties to contexts

Regularity: If $\Theta \vdash \Gamma$ 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

Structural Properties and Substitution of Types into Terms

- (Type Weakening of Terms) If $\Theta, \Theta' \vdash \Gamma \text{ ctx}$ and $\Theta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Contraction of Terms) If $\Theta, \alpha, \beta, \Theta' \vdash \Gamma \text{ ctx}$ and $\Theta, \alpha, \beta, \Theta'; \Gamma \vdash e : A$ then $\Theta, \beta, \alpha, \Theta'; \Gamma \vdash e : A$.
- (Type Substitution of Terms) If $\Theta, \alpha \vdash \Gamma \text{ ctx}$ and $\Theta \vdash A \text{ type}$ and $\Theta, \alpha; \Gamma \vdash e : B$ then $\Theta; [A/\alpha]\Gamma \vdash [A/\alpha]e : [A/\alpha]B$.

Structural Properties and Substitution for Term Variables

- (Weakening of Terms) If $\Theta \vdash \Gamma, \Gamma' \text{ ctx}$ and $\Theta \vdash B \text{ type}$ and $\Theta; \Gamma, \Gamma' \vdash e : A$ then $\Theta; \Gamma, y : B, \Gamma' \vdash e : A$
- (Contraction of Terms) If $\Theta \vdash \Gamma, y : B, z : C, \Gamma' \text{ ctx}$ 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 $\Theta \vdash \Gamma, x : A \text{ ctx}$ 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!)