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	$\mathsf{L}\ \langle\rangle$
false	$\mathsf{R}\ \langle\rangle$
if e then e' else e''	$\text{case}(e, \mathsf{L}\ \_ \to e', \mathsf{R}\ \_ \to e'')$

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

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

$$\frac{\Gamma, f : X \to Y, x : X \vdash e : Y}{\Gamma \vdash \mathrm{fun}_{X \to Y} f x.\, e : X \to Y} \textsc{Fix}$$

$$\frac{e' \rightsquigarrow e''}{(\mathrm{fun}_{X \to Y} f x.\, e)\, e' \rightsquigarrow (\mathrm{fun}_{X \to Y} f x.\, e)\, e''} \qquad \frac{}{(\mathrm{fun}_{X \to Y} f x.\, e)\, v \rightsquigarrow [\mathrm{fun}_{X \to Y} f x.\, e/f, v/x]e}$$

- 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

$$
\dfrac{\cdots \qquad \dfrac{\cdots \qquad \dfrac{\Delta \vdash \mathit{fact} : \mathtt{int} \to \mathtt{int} \qquad \dfrac{\vdots}{\Delta \vdash n - 1 : \mathtt{int}}}{\Delta \vdash \mathit{fact}(n-1) : \mathtt{int}}}{\Delta \vdash n \times \mathit{fact}(n-1) : \mathtt{int}}}{\dfrac{\overbrace{\Gamma, \mathit{fact} : \mathtt{int} \to \mathtt{int}, n : \mathtt{int}}^{\Delta} \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times \mathit{fact}(n-1) : \mathtt{int}}{\Gamma \vdash \text{fun}_{\mathtt{int} \to \mathtt{int}}\, \mathit{fact}\, n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \mathit{fact}(n-1) : \mathtt{int} \to \mathtt{int}}}
$$

A Bad Use of Recursion

$$\dfrac{\dfrac{}{f : 1 \to 0, x : 1 \vdash f : 1 \to 0} \qquad \dfrac{}{f : 1 \to 0, x : 1 \vdash x : 1}}{\dfrac{f : 1 \to 0, x : 1 \vdash f x : 0}{\cdot \vdash \mathrm{fun}_{1 \to 0} f x. f x : 1 \to 0}}$$

$$\begin{array}{rcl} (\mathrm{fun}_{1\to 0} f x. f x)\ \langle\rangle & \rightsquigarrow & [\mathrm{fun}_{1\to 0} f x. f x/f, \langle\rangle\ /x](f x) \\ & \equiv & (\mathrm{fun}_{1\to 0} f x. f x)\ \langle\rangle \\ & \rightsquigarrow & [\mathrm{fun}_{1\to 0} f x. f x/f, \langle\rangle\ /x](f x) \\ & \equiv & (\mathrm{fun}_{1\to 0} f x. f x)\ \langle\rangle \\ & \cdots & \end{array}$$

Numbers, More Safely

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

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \qquad \Gamma \vdash e_1 : X \qquad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \mathsf{iter}(e_0, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) : X}\ \mathbb{N}\mathsf{E}$$

$$\frac{e_0 \leadsto e_0'}{\mathsf{iter}(e_0, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) \leadsto \mathsf{iter}(e_0', \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2)}$$

$$\frac{}{\mathsf{iter}(\mathsf{z}, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) \leadsto e_1}$$

$$\frac{}{\mathsf{iter}(\mathsf{s}(v), \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) \leadsto [\mathsf{iter}(v, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2)/x]e_2}$$

Expressiveness of Gödel's T

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

$$\begin{aligned} n + m &\triangleq \mathrm{iter}(n, \mathrm{z} \to m, \mathrm{s}(x) \to \mathrm{s}(x)) \\ n \times m &\triangleq \mathrm{iter}(n, \mathrm{z} \to \mathrm{z}, \mathrm{s}(x) \to m + x) \\ \mathtt{pow}(n, m) &\triangleq \mathrm{iter}(m, \mathrm{z} \to \mathrm{s}(\mathrm{z}), \mathrm{s}(x) \to n \times x) \end{aligned}$$

- 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

$$\begin{aligned}
\mathsf{repeat} &: (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \\
\mathsf{repeat} &\triangleq \lambda f.\, \lambda n.\, \mathsf{iter}(n, \mathsf{z} \to id, \mathsf{s}(r) \to f \circ r)
\end{aligned}$$

$$\begin{aligned}
\mathsf{ack} &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{ack} &\triangleq \lambda m.\, \mathsf{iter}(m, \mathsf{z} \to (\lambda n.\, \mathsf{s}(n)), \mathsf{s}(r) \to \lambda n.\, \mathsf{repeat}\, r\, n\, (r\, 1))
\end{aligned}$$

- Proposition: $A(n, m) \triangleq \mathsf{ack}\, n\, m$
- 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

$$\frac{}{\Gamma \vdash [] : \mathsf{list}\, X}\ \textsc{ListNil} \qquad \frac{\Gamma \vdash e : X \qquad \Gamma \vdash e' : \mathsf{list}\, X}{\Gamma \vdash e :: e' : \mathsf{list}\, X}\ \textsc{ListCons}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{list}\, X \qquad \Gamma \vdash e_1 : Z \qquad \Gamma, x : X, r : Z \vdash e_2 : Z}{\Gamma \vdash \mathsf{fold}(e_0, [] \to e_1, x :: r \to e_2) : Z}\ \textsc{ListFold}$$

Data Structures: Lists

$$\frac{e_0 \rightsquigarrow e_0'}{e_0 :: e_1 \rightsquigarrow e_0' :: e_1} \qquad \frac{e_1 \rightsquigarrow e_1'}{v_0 :: e_1 \rightsquigarrow v_0 :: e_1'}$$

$$\frac{e_0 \rightsquigarrow e_0'}{\mathrm{fold}(e_0, [] \to e_1, x :: r \to e_2) \rightsquigarrow \mathrm{fold}(e_0', [] \to e_1, x :: r \to e_2)}$$

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

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

Some Functions on Lists

$$
\begin{aligned}
\mathsf{length} &: \mathsf{list}\,X \to \mathbb{N} \\
\mathsf{length} &\triangleq \lambda xs.\,\mathsf{fold}(xs, [] \to \mathsf{z}, x :: r \to \mathsf{s}(r)) \\[1em]
\mathsf{append} &: \mathsf{list}\,X \to \mathsf{list}\,X \to \mathsf{list}\,X \\
\mathsf{append} &\triangleq \lambda x.\,\lambda ys.\,\mathsf{fold}(xs, [] \to ys, x :: r \to x :: r) \\[1em]
\mathsf{map} &: (X \to Y) \to \mathsf{list}\,X \to \mathsf{list}\,Y \\
\mathsf{map} &\triangleq \lambda f.\,\lambda xs.\,\mathsf{fold}(xs, [] \to [], x :: r \to (f\,x) :: r)
\end{aligned}
$$

A Logical Perversity

- The Curry-Howard Correspondence tells us to think of *types as propositions*
- But what logical propositions do $\mathbb{N}$ or $\mathrm{list}\,X$, correspond to?
- The following biconditionals hold:
    - $1 \iff \mathbb{N}$
    - $1 \iff \mathrm{list}\,X$
    - $\mathbb{N} \iff \mathrm{list}\,X$
- So $\mathbb{N}$ is “equivalent to” truth?

A Practical Perversity

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

- 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

$$\begin{array}{llll} \text{Types} & A & ::= & \alpha \mid A \to B \mid \forall \alpha.\, A \\ \text{Terms} & e & ::= & x \mid \lambda x : A.\, e \mid e\, e \mid \Lambda \alpha.\, e \mid e\, A \end{array}$$

- We want to support *type polymorphism*
  - append : $\forall \alpha.\, \mathsf{list}\, \alpha \to \mathsf{list}\, \alpha \to \mathsf{list}\, \alpha$
  - map : $\forall \alpha.\, \forall \beta.\, (\alpha \to \beta) \to \mathsf{list}\, \alpha \to \mathsf{list}\, \beta$
- 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

$$\text{Type Contexts} \quad \Theta \quad ::= \quad \cdot \mid \Theta, \alpha$$

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha \text{ type}} \qquad \frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash B \text{ type}}{\Theta \vdash A \to B \text{ type}} \qquad \frac{\Theta, \alpha \vdash A \text{ type}}{\Theta \vdash \forall \alpha. A \text{ type}}$$

- Judgement $\Theta \vdash A$ 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

$$\text{Term Variable Contexts} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

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

- Judgement $\Theta \vdash \Gamma$ ctx 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

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

$$\frac{\Theta \vdash A \text{ type} \qquad \Theta; \Gamma, x : A \vdash e : B}{\Theta; \Gamma \vdash \lambda x : A.\, e : A \to B} \qquad \frac{\Theta; \Gamma \vdash e : A \to B \qquad \Theta; \Gamma \vdash e' : A}{\Theta; \Gamma \vdash e\, e' : B}$$

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

- 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 $\Theta, \Theta' \vdash \Gamma$ 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 of Typing

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 $\Theta, \Theta' \vdash \Gamma$ 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 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!)