Types and Logics

Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

Curry-Howard Isomorphism

Idea

$$\begin{aligned} \textit{types} &= \textit{formulæ} \\ \textit{programs} &= \textit{proofs} \end{aligned}$$

- Implication fragment of Propositional Logic and

  Simply-typed λ-calculus:

$$\frac{\pi \uplus \{x : \theta\} \vdash T : \theta'}{\pi \vdash \lambda x.T : \theta \to \theta'} (\to I) \qquad \frac{\pi \vdash T_1 : \theta \to \theta' \qquad \pi \vdash T_2 : \theta}{\pi \vdash (T_1\ T_2) : \theta'} (\to E)$$

- What is the role of the β-rule?

  $\Rightarrow$ simplification of proofs!

- Simplification of proofs = *cut elimination*

Conjunction

- Similar for the $(\wedge, \rightarrow)$ fragment:

$$\frac{\pi \vdash T_1 : \theta_1 \qquad \pi \vdash T_2 : \theta_2}{\pi \vdash \langle T_1, T_2 \rangle : \theta_1 \wedge \theta_2}(\wedge I)$$

$$\frac{\pi \vdash T : \theta_1 \wedge \theta_2}{\pi \vdash (\mathbf{pr}_1 \ T) : \theta_1}(\wedge E_1) \qquad \frac{\pi \vdash T : \theta_1 \wedge \theta_2}{\pi \vdash (\mathbf{pr}_2 \ T) : \theta_2}(\wedge E_2)$$

- we can now can *prove*, e.g.,

$$((A \wedge B) \rightarrow C) \rightarrow (A \rightarrow (B \rightarrow C))$$

Idea

Conjunctions = Pairs

Disjunction

- Can we *use* disjunction?

$$\frac{\pi \vdash T : \theta_1}{\pi \vdash \textbf{inr}\ T : \theta_1 \vee \theta_2}(\vee I_1) \qquad \frac{\pi \vdash T : \theta_2}{\pi \vdash \textbf{inr}\ T : \theta_1 \vee \theta_2}(\vee I_2)$$

$$\frac{\pi \vdash T : \theta_1 \vee \theta_2 \qquad \pi \vdash T_1 : \theta_1 \to \theta \qquad \pi \vdash T_2 : \theta_2 \to \theta}{\pi \vdash \textbf{case}\ T\ \textbf{of}\ \textbf{inl}(x) : (T_1\ x) \mid \textbf{inr}(x) : (T_2\ x) : \theta} \vee E$$

- this is *the canonical* way of dealing with $\vee$

  $\Rightarrow$ the **inl** and **inr** should be tagged by the disjunction.

Idea

Disjunctions = Variants

What Happened to Negation?

- Is there *any use* for negation?

  $\Rightarrow$ not really (for program construction)

  $\Rightarrow$ can be introduced in the logic

- weak negation via a constant for *false* ($\bot$):

$$\neg\theta \equiv \theta \rightarrow \bot$$

- this only allows *intuitionistic* proofs

  $\Rightarrow$ there is a *witness* for every valid formula

  $\Rightarrow$ not true in *classical* logic ($\vdash \theta \vee \neg\theta$)

Type Variables and Quantification

Idea

We extend the grammar for types with type variables

$$\tau ::= \iota \mid \tau \to \tau \mid \alpha$$

- what do the free variables α stand for?

  $\Rightarrow$ only closed terms are valid types!

- we need quantifiers:

$$\tau ::= \iota \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\tau \mid \exists\alpha.\tau$$

($\forall\alpha.\tau$ universal type $\exists\alpha.\tau$ existential type)

Universal Types

Idea

A universal type can be specialized

$\Rightarrow$ *by substituting a type for the quantified variable*

- How is such a substitution manifested in a program/proof?

  $\Rightarrow$ similar to abstraction/application

  except this time the actual parameter is a *type*

$$\frac{\pi, \alpha \vdash T : \tau}{\pi \vdash \Lambda\alpha.T : \forall\alpha.\tau}(\forall I) \qquad \frac{\pi \vdash T : \forall\alpha.\tau}{\pi \vdash (T\ \tau') : [\tau'/\alpha]\tau}(\forall E)$$

  $\Rightarrow$ needs a generalization of *type assignments*

System F [Girard 1972]

What is this good for?

$\Rightarrow$ a *clean* way to understand complex types

Definition (System F)

Syntax: terms (T) and types (τ):

$$\begin{array}{rcl} T & ::= & x \mid \lambda x.T \mid (T\ T) \mid \Lambda\alpha.T \mid (T\ \tau) \\ \tau & ::= & \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \end{array}$$

Reductions:

$$(\lambda x.T\ T') \to [T'/x]T \qquad (\Lambda\alpha.T\ \tau) \to [\tau/\alpha]T$$

How good is F?

- polymorphic identity:

$$\Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha$$

- unlike simply typed λ-calculus, it can *type* $\lambda x.(x\ x)$:

$$\lambda x : \forall\alpha.\alpha \to \alpha.\,(x\ \forall\alpha.\alpha \to \alpha\ x) : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

- can it assign a type to Ω?

  $\Rightarrow$ NO! we can still prove strong normalization

Church-style Encodings

Idea

We encode standard data types using Church-style terms

$\Rightarrow$ and still assign types in system F

- Booleans:

$$
\begin{aligned}
\mathit{true} &= \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x : \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathit{false} &= \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y : \forall\alpha.\alpha \to \alpha \to \alpha \\
\mathit{if} &= \lambda b : \forall\alpha.\alpha \to \alpha \to \alpha.\lambda x : \alpha.\lambda y.\alpha.(b\ \alpha\ x\ y) : \alpha
\end{aligned}
$$

$$
\begin{aligned}
\mathit{not} &= \lambda b : \forall\alpha.\alpha \to \alpha \to \alpha.\Lambda\beta.\lambda t : \beta.\lambda f : \beta.(b\ \beta\ f\ t) \\
&\quad : (\forall\alpha.\alpha \to \alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha \to \alpha)
\end{aligned}
$$

Encoding of Products

- Pairs:

$$
\begin{aligned}
\mathit{pair} &= \lambda x : \alpha.\lambda y : \beta.\Lambda\gamma.\lambda z : \alpha \to \beta \to \gamma.(z\ x\ y) \\
&\quad : \alpha \to \beta \to (\forall\gamma.\alpha \to \beta \to \gamma) \\
\mathit{proj}_1 &= \lambda p : \mathit{Pair}.(p\ \alpha\ (\lambda x : \alpha.\lambda y : \beta.x)) : \alpha \\
\mathit{proj}_2 &= \lambda p : \mathit{Pair}.(p\ \beta\ (\lambda x : \alpha.\lambda y : \beta.y)) : \beta
\end{aligned}
$$

- Reductions:

$$
\begin{aligned}
\mathit{proj}_1(\mathit{pair}\ u\ v) &= (\Lambda\gamma.\lambda z : \gamma \to \alpha \to \beta.z\ u\ v)\ \alpha\ (\lambda x : \alpha.\lambda y : \beta.x) \\
&= (\lambda z : \alpha \to \beta \to \alpha.z\ u\ v)\ (\lambda x : \alpha.\lambda y : \beta.x) \\
&= (\lambda x : \alpha.\lambda y : \beta.x)\ u\ v \\
&= (\lambda y : \beta.u)\ v \\
&= u
\end{aligned}
$$

Encoding of Natural Numbers

- Natural Numbers (Church numerals):

$$\mathit{Nat} = \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$$

$$\begin{array}{rcl} \mathit{zero} & = & \Lambda\alpha.\lambda z : \alpha.\lambda s : \alpha \to \alpha.x \\ \mathit{succ} & = & \lambda n : \mathit{Nat}.\Lambda\alpha.\lambda z : \alpha.\lambda s : \alpha \to \alpha.(s\ (n\ \alpha\ z\ s)) \end{array}$$

- Iterator:

$$\mathit{iter} \quad = \quad \lambda n : \mathit{Nat}.\lambda u : \beta.\lambda v : \beta \to \beta.(n\ \beta\ u\ v)$$

Encoding of Polymorphic Lists, Trees, . . .

- Lists (of β's):

$$\mathit{List}(\beta) = \forall\alpha.\alpha \to (\beta \to \alpha \to \alpha) \to \alpha$$

- Binary Trees (β's in nodes and γ's in leaves):

$$\mathit{BinTree}(\beta, \gamma) = \forall\alpha.(\gamma \to \alpha) \to (\beta \to \alpha \to \alpha \to \alpha) \to \alpha$$

- Trees of branching type β:

$$\beta\text{-}\mathit{Tree} = \forall\alpha.\alpha \to ((\beta \to \alpha) \to \alpha) \to \alpha$$

Existential Types

What does $\exists\alpha.\tau$ stand for?

Idea

Allow to hide the type used e.g., in an implementation

$\Rightarrow$ *types used locally in modules/classes*

- How do elements of $\exists\alpha.\tau$ look like?

$$\{\tau', T\} : \exists\alpha.\tau \qquad \textit{if} \qquad T : [\tau'/\alpha]\tau$$

$\Rightarrow$ the type τ' is *hidden* from users of $\exists\alpha.\tau$

Example (ML-like modules)

- We want a counter module with signature:

$$\exists Count.\langle Count, Count \to Nat, Count \to Count \rangle$$

$\Rightarrow$ with operations *zero*, *get*, and *inc*

- and we can have an implementation:

$$\{Nat, \langle 0, \lambda x.x, \lambda x.x + 1\rangle\}$$
$$\{NatList, \langle nil, \lambda x.length\ x, \lambda x.0 :: x\rangle\}$$

- Given a counter c we would like to be able to write:

$$(\mathbf{Pr}_2\, c)((\mathbf{Pr}_3\, c)(\mathbf{Pr1}_1\, c)) = 1 : Nat$$

$\Rightarrow$ we need to associate the *implementation* with the *type*

idea (fictional syntax!): $\mathbf{open}\, T : \exists\beta.\tau'\ \mathbf{as}\{\alpha, x\}\ \mathbf{in}\, T'$

Typing Rules and Reductions

- Typing rules:

$$\frac{\pi \vdash T : [\tau'/\alpha]\tau}{\pi \vdash \{\tau', T\} : \exists\alpha.\tau}\ (\exists I) \qquad \frac{\pi \vdash T : \exists\alpha.\tau \qquad \pi \uplus \{\alpha, x : \tau\} \vdash T' : \tau'}{\pi \vdash \textbf{open}\ T\ \textbf{as}\{\alpha, x\}\ \textbf{in}\ T' : \tau'}\ (\exists E)$$

- Reduction:

$$(\textbf{open}\ T\ \textbf{as}\{\alpha, x\}\ \textbf{in}\ T')\ \{\tau, T\} \rightarrow [\tau/\alpha, T/x]T'$$

Coding $\exists\alpha.\tau$ as an Universal Type

- the existential type $\exists\alpha.\tau$ can be coded:

$$\exists\alpha.\tau = \forall\beta.(\forall\alpha.\tau \to \beta) \to \beta$$

- operations ($T : [\theta/\alpha]\tau$):

$$\begin{aligned} \{\theta, T\} &= \Lambda\beta.\lambda x : \forall\alpha.\tau \to \beta.x\ \theta\ T \\ \textbf{open}\, T_1\, \textbf{as}\{\alpha, x\}\, \textbf{in}\, T_2 : \tau' &= T_1\ \tau'\ (\Lambda\alpha.\lambda x : \tau.T_2) \end{aligned}$$

- reduction:

$$\begin{aligned} &\textbf{open}\{\theta, T\}\, \textbf{as}\{\alpha, x\}\, \textbf{in}\, T' : \tau' \\ &\quad = (\Lambda\beta.\lambda x : \forall\alpha.\tau \to \beta.x\ \theta\ T)\ \tau'\ (\Lambda\alpha.\lambda x : \tau.T') \\ &\quad = (\lambda x : \forall\alpha.\tau \to \tau'.x\ \theta\ T)\ (\Lambda\alpha.\lambda x : \tau.T') \\ &\quad = (\Lambda\alpha.\lambda x : \tau.T')\ \theta\ T \\ &\quad = [\theta/\alpha, T/x]T' \end{aligned}$$

Types as Parameters and Kinds

So far we can pass types as parameters to terms

what would happen if we parametrized types themselves?

Idea

Add a level of kinds: $K ::= * \mid K \to K$ *(* $*$ *is a proper type)*

- Now type (functions) can be applied on types:

  Pair is now properly a "$* \to * \to *$" type constructor

  ...similarly *List* : $* \to *$, *BinTree* : $* \to * \to *$, etc.

- We can construct System F_ω

  $\Rightarrow (\Box, \Box)$ polymorphism (in addition to $(\Box, *)$ in System F)

  $\Rightarrow$ still strongly normalizing

- Can we *repeat* the construction again?

  $\Rightarrow$ no: adding 3rd level = inconsistent (logical) system

(Almost) Dependent Types

OK—is there $(*, \Box)$ polymorphism (and what would that be?)

Idea

Parametrize types by values.

- for example $\lambda x : Nat : Nat^n \quad \sim \quad \textbf{array}[n]\,\textbf{of}\,Nat$
- true dependent types are proper (saturated) types!

  and the above is a type constructor (so a coercion is needed)

Summary

- (Intuitionistic) Logics give natural explanations of many (if not most) constructs in programming languages

  $\Rightarrow$ differences = variant syntax

- Several *surprising* results:

  $\Rightarrow$ provable termination for powerful languages (F_ω)

- Basis for industry-strength languages:

  $\Rightarrow$ SML/NJ, Haskell

  $\Rightarrow$ polymorphic extensions of JAVA