Type Inference Algorithm(s)

Adalynn Taylor

Questions

1. What is the most general type *system* in the Lambda Cube?
2. What type systems do Algorithm W apply to?
3. What's the problem with type inference for more general systems?

About Me

- Senior Undergraduate, Math Major. Advisor for Honors Thesis: Cartwright.
- Mathematical Logic, Formalization, Formal Methods
    - (hence my topic choice)


I LOVE LEAN

LEAN

THEOREM PROVER

(^ specifically, this Lean ^)

APL

Outline

- Overview
- History of Type Theory
  - Russell
  - Martin-Lof
  - Coquand
  - Lambda Cube
- Algorithms
  - Algorithm W
  - Unification
  - Higher? (Answer: no)
- Applications
  - Type Systems, Proof Assistants
- Implementations?
- Open Issues

Overview

A **type**, roughly speaking, is a collection of objects

Type theory is the study of how rule systems governing these types interact.

Dependent type systems have types that can depend on terms

(think of int[3])

MGU (Most General Unifier):

A polymorphic type

Principia Mathematica

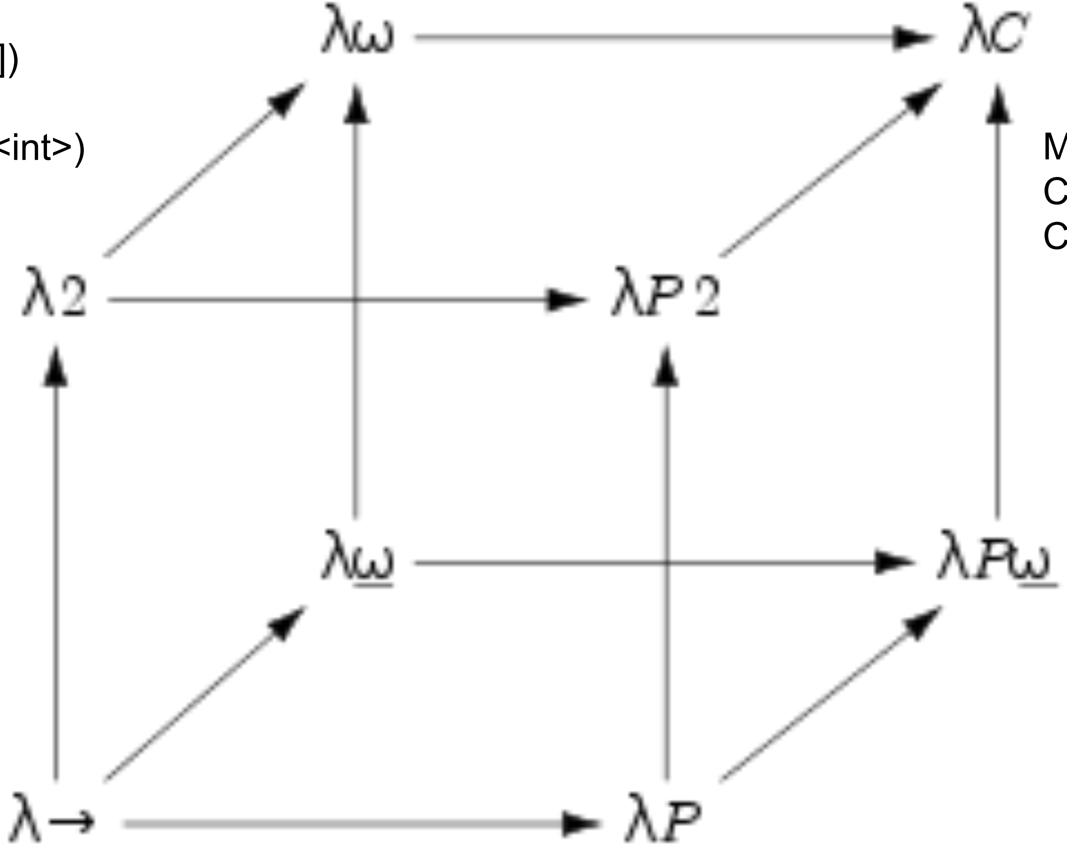
Functional Programming Languages

Proof assistants

History

- *Principia Mathematica*
  - Theory of Types - was created to resolve Russell’s paradox.
  - Later *ramified*, i.e. distinction between real and apparent variables collapsed.
- Martin-Lof type theory
  - Dependent Types
- Calculus of Constructions, Coquand and Huet
- Barendregt's lambda cube

X: Dependent Types (think int[3])
Y: Polymorphism (think f<T>)
Z: Type Operators (think vector<int>)



Most general:
Calculus of
Constructions

Least General: Simple Types

Algorithms: W (not imperative) and J (imperative)

Algorithm W

$$\frac{x : \sigma \in \Gamma \quad \tau = \mathit{inst}(\sigma)}{\Gamma \vdash_W x : \tau, \emptyset} \quad [\mathtt{Var}]$$

$$\frac{\begin{array}{ll} \Gamma \vdash_W e_0 : \tau_0, S_0 & S_0\Gamma \vdash_W e_1 : \tau_1, S_1 \\ \tau' = \mathit{newvar} & S_2 = \mathsf{mgu}(S_1\tau_0,\ \tau_1 \rightarrow \tau') \end{array}}{\Gamma \vdash_W e_0\ e_1 : S_2\tau', S_2 S_1 S_0} \quad [\mathtt{App}]$$

$$\frac{\tau = \mathit{newvar} \quad \Gamma,\ x : \tau \vdash_W e : \tau', S}{\Gamma \vdash_W \lambda\, x\ .\ e : S\tau \rightarrow \tau', S} \quad [\mathtt{Abs}]$$

$$\frac{\Gamma \vdash_W e_0 : \tau, S_0 \quad S_0\Gamma,\ x : \overline{S_0\Gamma}(\tau) \vdash_W e_1 : \tau', S_1}{\Gamma \vdash_W \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau', S_1 S_0} \quad [\mathtt{Let}]$$

Algorithm J

$$\frac{x : \sigma \in \Gamma \quad \tau = \mathit{inst}(\sigma)}{\Gamma \vdash_J x : \tau} \quad [\mathtt{Var}$$

$$\frac{\Gamma \vdash_J e_0 : \tau_0 \quad \Gamma \vdash_J e_1 : \tau_1 \quad \tau' = \mathit{newvar} \quad \mathit{unify}(\tau_0,\ \tau_1 \rightarrow \tau')}{\Gamma \vdash_J e_0\ e_1 : \tau'} \quad [\mathtt{App}$$

$$\frac{\tau = \mathit{newvar} \quad \Gamma,\ x : \tau \vdash_J e : \tau'}{\Gamma \vdash_J \lambda\, x\ .\ e : \tau \rightarrow \tau'} \quad [\mathtt{Abs}$$

$$\frac{\Gamma \vdash_J e_0 : \tau \qquad \Gamma,\ x : \bar{\Gamma}(\tau) \vdash_J e_1 : \tau'}{\Gamma \vdash_J \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau'} \quad [\mathtt{Let}$$

Huh?

Declarative Rule System

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash_D x:\sigma} \quad [\mathtt{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\mathtt{App}]$$

$$\frac{\Gamma, x:\tau \vdash_D e:\tau'}{\Gamma \vdash_D \lambda x\,.\,e : \tau \to \tau'} \quad [\mathtt{Abs}]$$

$$\frac{\Gamma \vdash_D e_0 : \sigma \qquad \Gamma, x:\sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau} \quad [\mathtt{Let}]$$

$$\frac{\Gamma \vdash_D e:\sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e:\sigma} \quad [\mathtt{Inst}]$$

$$\frac{\Gamma \vdash_D e:\sigma \quad \alpha \notin \mathrm{free}(\Gamma)}{\Gamma \vdash_D e : \forall \alpha\,.\,\sigma} \quad [\mathtt{Gen}]$$

Expressions

$$\begin{array}{lcll} e & = & x & \text{variable} \\ & | & e_1\ e_2 & \text{application} \\ & | & \lambda\, x\,.\, e & \text{abstraction} \\ & | & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 & \end{array}$$

Types

$$\begin{array}{llcll} \text{mono} & \tau & = & \alpha & \text{variable} \\ & & | & C\ \tau \dots \tau & \text{application} \\ \text{poly} & \sigma & = & \tau & \\ & & | & \forall\, \alpha\,.\, \sigma & \text{quantifier} \end{array}$$

Context and Typing

$$\begin{array}{llcl} \text{Context} & \Gamma & = & \epsilon\ (\mathtt{empty}) \\ & & | & \Gamma,\ x:\sigma \\ \text{Typing} & & = & \Gamma \vdash e:\sigma \end{array}$$

Free Type Variables

$$\begin{array}{lcl} \text{free}(\ \alpha\ ) & = & \{\alpha\} \\ \text{free}(\ C\ \tau_1 \dots \tau_n\ ) & = & \bigcup\limits_{i=1}^{n} \text{free}(\ \tau_i\ ) \\ \text{free}(\ \Gamma\ ) & = & \bigcup\limits_{x:\sigma \in \Gamma} \text{free}(\ \sigma\ ) \\ & & \\ \text{free}(\ \forall\, \alpha\,.\, \sigma\ ) & = & \text{free}(\ \sigma\ )\ -\ \{\alpha\} \\ \text{free}(\ \Gamma \vdash e:\sigma\ ) & = & \text{free}(\ \sigma\ )\ -\ \text{free}(\ \Gamma\ ) \end{array}$$

Algorithms **W** and **J** correspond to Hindley-Miller Type Systems.

Not *exactly* on the lambda-cube

Somewhere between Simply Typed and System F (aka lambda2)

Unification

Both algorithms W and J rely on *unification* and *finding the most general type.*

Unification, to simplify, is solving equations based on symbolic expressions.

First-order unification has an algorithm [Robinson; Martelli, Montanari]

But

Unification - Higher Order

Higher order unification, however, *is not decidable.*

F

What does that *mean?* Why aren’t there more?

It means that type systems like F, and *especially* proof assistants, need to have *some* explicit type annotation *somewhere*, or otherwise there is no most general unifier!

```
lemma congr {α: Sort u} {β: Sort v} {f g: α → β} {a b: α}: f = g → a = b → f a = g b :=
  λh₁: f = g,
    λh₂: a = b,
      λφ: β → Prop,
        iff.intro
          (λhpfa: φ (f a), (h₂ (λa', φ (g a'))).mp ((h₁ (λf', φ (f' a))).mp hpfa)) --Leibnizian
          (λhpgb: φ (g b), (h₂ (λb', φ (f b'))).mpr ((h₁ (λg', φ (g' b))).mpr hpgb)) --Chicanery
```

Why?

Applications

HM is used as the basis for a few functional programming language's type systems, of course with extensions.

Proof assistants

- Formalization of Mathematics
- Formal Methods (Software Verification!)

Linguistics (go see formal semantics of natural languages for THAT connection)

Questions (Revisited)

1. What is the most general type *system* in the Lambda Cube?
2. What type systems do Algorithm W apply to?
3. What's the problem with type inference for more general systems?

References

https://plato.stanford.edu/entries/type-theory/

https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system