Languages & Translators

LINGI2132

The Hindley-Milner Type System

Nicolas LAURENT
Université catholique de Louvain

The Hindley-Milner Type System

- Hindley-Mindler is a type system for
  the *polymorphic lambda calculus*, aka *System F*

- Relevant in practice: it underlies the Haskell and ML type systems.

- Can be extended in various useful ways.

Lambda Calculus

- Hindley-Mindler is a type system for
  the *polymorphic lambda calculus*, aka *System F*

- Lambda Calculus

  `e ::= x | (λx.e) | (e e)`

`x`	`Variable`
`(λx.e)`	`Abstraction`
`(e e)`	`Application`

- Simply-Typed Lambda Calculus (λ→)

  `e ::= x | (λx:`τ`.e) | (e e) | c`

τ	`Parameter Type`
`c`	`Constant`

Lambda Calculus

- Simply-Typed Lambda Calculus (λ→)

  `e ::= x | (λx:τ.e) | (e e) | c`

  - we need a set of base types, e.g. `B = {a, b}`
  - our types are `a`, `b`, and any function type: e.g. `a → (a → b)`
  - why are constants necessary? (no Church encoding)

Lambda Calculus

- *Polymorphic* Lambda Calculus
  - idea: type the original lambda calculus - **without type annotations**
  - `e ::= x | (λx.e) | (e e) | let x = e in e`
  - we will explain the need for `let` later

A Type System?

- A formal system (formalism) that determines
  - if any expression in the language is well-typed
  - the type of any such expression
  - `e ::= x | (λx:τ.e) | (e e) | c`
  - ill-typed: `((λ x:a . x) c1)` where `c1` has type `b`

A Type System?

- Desirable properties
    - **Decidability**: we can make a decision.
    - **Soundness**: everything we can prove is true.
    - **Completeness**: we can prove everything that is true.
- Hard Limits: Gödel's incompleteness theorem
    - No sound system of axioms describing natural number arithmetic can be complete.
    - Pragmatic considerations

Type System for λ→

- Γ = context, a set of type bindings (assignments, assumptions)
- ⊢ = type judgement
- σ, τ denote types

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \text{ (1)} \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T} \text{ (2)}$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)} \text{ (3)} \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau} \text{ (4)}$$

```
((λ x:a . x) c)
```

https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus

Towards Hindley-Milner

- functions can have many types!

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

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

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

```
(λ x . x)
```

https://en.wikipedia.org/wiki/Hindley–Milner_type_system

Polymorphism

- What is the type of `(λ x . x)` ?
  - identity function
  - `∀ α . α → α`
  - polymorphic type

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

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

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

mono	τ	$=$	α	variable
		~~$\mid$~~	~~$C\ \tau \ldots \tau$~~	~~application~~
		$\mid$	$\tau \to \tau$	abstraction
poly	σ	$=$	τ	
		$\mid$	$\forall\, \alpha\ .\ \sigma$	quantifier

Why do we need polymorphism?

```
((λid . ((foo (id 1)) (id 's')))
 (λx . x))
```

Generalization

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

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

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

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

```
⊢ (λ x . x) : α → α          (Abs)

⊢ (λ x . x) : ∀ α . α → α    (Gen)
```

Which are variables are bound (unfree)?

- ∀-quantified variables
- variables appearing in function types
  - ... that never appear on their own

Instantiation

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

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

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

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

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

⊢ (λ x . x) : α → α (Abs)

⊢ (λ x . x) : ∀ α . α → α (Gen)

⊢ (λ x . x) : int → int (Inst)

Why do we need polymorphism?

```
((λid . ((foo (id 1)) (id 's')))
 (λx . x))
```

Let-polymorphism

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

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

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

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

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

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

Isn't `let x = e1 in e2` the same as `((λx.e2) e1)` ???

Typing the real polymorphic lambda calculus is **undecidable**.

```
((λid . ((foo (id 1)) (id 's')))
 (λx . x))
```

```
let id = (λx . x) in
  ((foo (id 1)) (id 's'))
```

Type System vs Typing Algorithm

- The inference rules give the *semantics* of the type system.
- The rules can be used to prove statements about types.
  = Obtain typing judgements.
- These statements are true (because the system is *sound*).
- What's not given: **how** to prove the statements.
  Which rules to use and when!
- For this you need a typing algorithm.
    - For Hindley-Milner: Algorithm J, **Algorithm W**.

Summary

- Formal type systems usually works through the use of inference rules, which are used to derive type judgements (⊢), supported by a set of type bindings (Γ).
- Hindley-Milner is a type system for the polymorphic lambda calculus.
    - Only 6 rules, enables the use of polymorphic (function) types.
    - Relevant in practice: it is the theory being some language's typing algorithms.
- Decidability / Soundness / Completeness
- The idea of *inference rules* has great practical applications.
    - Inference rule = premises → conclusions, which may serve as further premises
    - Typing can be seen as a dependency network between terms (cf. Uranium)

Next Time

Semantic Analysis with Uranium