Polymorphic Typing (I)

Sjaak Smetsers

27 February 2025

Semantic Analysis

- Semantic analysis is more than type checking
  - Happens between parsing and code generation
  - Builds/Uses a symbol table, mapping identifiers to their declaration
- Semantic analysis may include
  - Type inference, Type checking
  - Strictness analysis
  - Uniqueness analysis
  - Reachability analysis
  - Dataflow analysis

Semantics

Syntax: Grammatical structure

Semantics: Meaning

- Operational How the effect of a program is produced.
  - Natural Semantics
  - Structural Operational Semantics
- Denotational What the effect of a program is.
- Axiomatic Which properties a program has.

What is a Type?

A type is a description of a *set of values* (and a set of allowed operations on those values).

Examples

Int is the set of all integers
Float is the set of all floats
Bool is the set $\{$**true**, **false**$\}$

More examples

List Int is the set of all lists of integers
List is a *type constructor*: A mapping from types to types
Foo, in Java, is the set of all objects of class **Foo**
Int $\rightarrow$ **Int** is the set of functions mapping an integer to an integer.
E.g., increment, decrement, and many others

Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)
- Basis for type soundness theorems: ‘any well-typed program cannot produce run-time errors (of some specified kind)’.
- Can decouple specification of typing aspects of a language from algorithmic concerns: the formal type system can define typing independently of particular implementations of type-checking algorithms.

Typing

Why?

- Safety
- Efficiency
- (Type driven) Development

What?

- Formally: Programs of type τ compute values of type τ
- Intuitively: prevents us from calculating 3 Volt + 2 Ampère

How?

- Type checking
- Type inference

λ-calculus: Syntax

$$
\begin{array}{rcl}
M & ::= & x \\
 & | & M_1 M_2 \\
 & | & \lambda x.M
\end{array}
$$

In Haskell

```
data Lambda = Var String
            | App Lambda Lambda
            | Abs String Lambda
```

λ-calculus: Semantics

Standard operational semantics

- β-reduction, based on substitution

$$(\lambda x.M)N \quad \rightarrow_\beta \quad M[x \mapsto N]$$

- Reduction strategy indicates redexes
- Gives rise to the notion of nomalization and laziness

λ-calculus: Ingredients for the Type System

Types

$b \in B$ (base types)

$\sigma ::= b \mid \sigma_1 \to \sigma_2$

Environments

$\Gamma : \mathit{Variables} \to \sigma$

Example: $\Gamma = [\langle x, \mathit{int} \rangle, \langle y, \mathit{bool} \rangle]$

Notation: $x{:}\mathit{int}, y{:}\mathit{bool}$

Typing judgements

$\Gamma \vdash M : \sigma$

this should be read as M has type σ in context Γ

Derivation Rules

Derivations (proofs) are trees made up from gluing together derivation rules

$$\frac{\vdash A \qquad \vdash B \qquad \vdash C}{\vdash D}$$

A derivation rule can be read in two ways

Top-down: If we have proofs for A, B and C then we have a proof for D

Bottom-up: To prove D we have to prove A, B and C

The Type System $\lambda^{\rightarrow}$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textit{(Variable)}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}\ (\to\textit{-Elimination})$$

$$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau}\ (\to\textit{-Introduction})$$

$\Gamma, x{:}\sigma$ stands for "Γ extended by $x{:}\sigma$". Formally:

$$\{y{:}t \mid y{:}t \in \Gamma, y \neq x\} \cup \{x{:}\sigma\}$$

Syntax of SL

$$
\begin{array}{lcl}
e & ::= & x \text{ (variables)} \\
 & | & \lambda x.e \\
 & | & e_1 e_2 \\
 & | & \textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e \\
 & | & e_1 \; op \; e_2 \\
 & | & i \text{ (integers)} \\
 & | & b \text{ (booleans)} \\
op & ::= & + \mid \leq \mid \&\& \\
 & & \\
\sigma & ::= & \sigma_1 \to \sigma_2 \\
 & | & int \\
 & | & bool
\end{array}
$$

Expressions of the forms

- $\lambda x.e$
- i
- **True**, **False**

are called values.

Types are
int, $bool$,
$int \to int$,
$int \to (int \to int)$,
$(int \to int) \to int$,
$(bool \to int) \to (int \to bool)$,
$\ldots$

Type Derivation Rules for SL$^{\rightarrow}$

$$\frac{b \in \{\mathbf{True}, \mathbf{False}\}}{\Gamma \vdash b : \mathit{bool}}\ (\mathit{Bool}) \qquad \frac{i \in \{\ldots, -1, 0, 1, \ldots\}}{\Gamma \vdash i : \mathit{int}}\ (\mathit{Int})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash e_1 + e_2 : \mathit{int}}\ (+)$$

$$\frac{\Gamma \vdash e_1 : \mathit{bool} \qquad \Gamma \vdash e_2 : \mathit{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \mathit{bool}}\ (\&\&)$$

$$\frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : \mathit{int}}{\Gamma \vdash e_1 \leq e_2 : \mathit{bool}}\ (\leq)$$

$$\frac{\Gamma \vdash e_c : \mathit{bool} \qquad \Gamma \vdash e_t : \sigma \qquad \Gamma \vdash e_e : \sigma}{\Gamma \vdash \mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e : \sigma}\ (\mathit{If})$$

Type Derivation Rules for $\mathsf{SL}^{\rightarrow}$ (2)

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (\mathit{Var})$$

$$\frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}\ (\mathit{App})$$

$$\frac{\Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \rightarrow \tau}\ (\mathit{Abs})$$

Example

$$
\dfrac{
  \dfrac{
    \dfrac{x{:}int \in \Gamma_{xy}}{\Gamma_{xy} \vdash x : int}
    \qquad
    \dfrac{y{:}int \in \Gamma_{xy}}{\Gamma_{xy} \vdash y : int}
  }{\Gamma_{xy} \vdash x \leq y : bool}
  \qquad
  \dfrac{y{:}int \in x{:}int, y{:}int}{x{:}int, y{:}int \vdash y : int}
  \qquad
  \dfrac{x{:}int \in \Gamma_{xy}}{\Gamma_{xy} \vdash x : int}
}{
  \dfrac{x{:}int, y{:}int \vdash \textbf{if } x \leq y \textbf{ then } x \textbf{ else } y : int}{
    \dfrac{x{:}int \vdash \lambda y.\textbf{if } x \leq y \textbf{ then } x \textbf{ else } y : int \to int}{\vdash \lambda x.\lambda y.\textbf{if } x \leq y \textbf{ then } x \textbf{ else } y : int \to (int \to int)}
  }
}
$$

$$
\frac{\Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ (Abs)
$$

$$
\frac{\Gamma \vdash e_c : bool \qquad \Gamma \vdash e_t : \sigma \qquad \Gamma \vdash e_e : \sigma}{\Gamma \vdash \textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e : \sigma}\ (If)
$$

$$
\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \leq e_2 : bool}\ (\leq)
\qquad\qquad
\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var)
$$

Common (Typing) Problems

Type checking

Given M and σ, is it the case that $\vdash M : \sigma$?

Type inference aka type reconstruction

Given M, is there a type σ such that $\vdash M : \sigma$?

Type inhabitation

Given σ, is there a term M such that $\vdash M : \sigma$?

Operational Semantics for SL (numbers)

$$\frac{e_1 \to e_1'}{e_1 + e_2 \;\to\; e_1' + e_2} \;\textit{(+ left)} \qquad\qquad \frac{e_2 \to e_2'}{v + e_2 \;\to\; v + e_2'} \;\textit{(+ right)}$$

$$\frac{v_1 \in i \qquad v_2 \in i \qquad n = [\![v_1]\!] + [\![v_2]\!] \in \mathbb{Z}}{v_1 + v_2 \;\to\; \overline{n}} \;\textit{(+ eval)}$$

We don't have rules for values.

$$\frac{?}{\textbf{True} \to ?} \;\textit{(True?)} \qquad\qquad \frac{?}{i \to ?} \;\textit{(int?)}$$

Operational Semantics for SL (if)

$$\frac{e_c \to e_c'}{\textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e \;\to\; \textbf{if } e_c' \textbf{ then } e_t \textbf{ else } e_e}\ (\mathit{If})$$

$$\frac{}{\textbf{if True then } e_t \textbf{ else } e_e \;\to\; e_t}\ (\mathit{Then})$$

$$\frac{}{\textbf{if False then } e_t \textbf{ else } e_e \;\to\; e_e}\ (\mathit{Else})$$

Operational Semantics for SL (lambda)

$$\frac{e_f \rightarrow e'_f}{e_f e_a \ \rightarrow \ e'_f e_a} \ \textit{(App)}$$

$$\frac{}{(\lambda x.e_b)e_a \ \rightarrow \ e_b[x \mapsto e_a]} \ \textit{(Redex)}$$

No rules for abstractions and variables.

$$\frac{?}{\lambda x.e \rightarrow \ ?} \ \textit{(Abs?)} \qquad \frac{?}{x \ \rightarrow \ ?} \ \textit{(Var?)}$$

Examples

$$
\begin{array}{ll}
 & (\lambda x.\lambda y.\textbf{if } x \le y \textbf{ then } x \textbf{ else } y)5\ 7 \\
\rightarrow & (\lambda y.\textbf{if } 5 \le y \textbf{ then } 5 \textbf{ else } y)7 \\
\rightarrow & \textbf{if } 5 \le 7 \textbf{ then } 5 \textbf{ else } 7 \\
\rightarrow & \textbf{if True then } 5 \textbf{ else } 7 \\
\rightarrow & 5 \\
 & \text{and we're stuck}
\end{array}
$$

$$
\begin{array}{ll}
 & (\lambda x.\lambda y.\textbf{if } x + y \textbf{ then } x \textbf{ else } y)5\ 7 \\
\rightarrow & (\lambda y.\textbf{if } 5 + y \textbf{ then } 5 \textbf{ else } y)7 \\
\rightarrow & \textbf{if } 5 + 7 \textbf{ then } 5 \textbf{ else } 7 \\
\rightarrow & \textbf{if } 12 \textbf{ then } 5 \textbf{ else } 7 \\
 & \text{and we're stuck}
\end{array}
$$

Type-safety: well-typed terms don't get stuck on non-values

Towards Polymorphism

Consider this program

$$\begin{array}{ll} & (\lambda f.\textbf{if } (f \textbf{ True}) \textbf{ then } (f\ 5) \textbf{ else } 7)(\lambda x.x) \\ \rightarrow & \ldots \\ \rightarrow & 5 \end{array}$$

In a type derivation, which type do we give f?

$$\frac{f : \mathit{bool} \rightarrow \mathit{bool} \vdash \ldots \qquad f : \mathit{int} \rightarrow \mathit{int} \vdash \ldots}{\vdash (\lambda f.\textbf{if } (f \textbf{ True}) \textbf{ then } (f\ 5) \textbf{ else } 7)(\lambda x.x) : \mathit{int}}$$

The problem

- Semantically this program seems to be ok
- We cannot type it

Polymorphism

From the greek "poly" (many) "morphe" (form)

Polymorphic type system: one variable can have many types

The identity function $\lambda x.x$ has many types

$$\mathit{int} \to \mathit{int}$$
$$\mathit{bool} \to \mathit{bool}$$
$$(\mathit{int} \to \mathit{int}) \to (\mathit{int} \to \mathit{int})$$
$$(\mathit{bool} \to \mathit{int} \to \mathit{bool}) \to (\mathit{bool} \to \mathit{int} \to \mathit{bool})$$
$$\ldots$$

But if we bind it to a variable, we must choose a single type

$$\textit{for any concrete type } \tau,\ \lambda x.x : \tau \to \tau$$

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

The Polymorphic Lambda Calculus λ_2 (aka *System F*)

Polymorphic types

$$\begin{array}{rcl} b & \in & B \text{ (base types)} \\ \alpha & \in & V \text{ (type variables)} \\ \sigma & ::= & b \mid \alpha \mid \sigma \to \sigma \mid \forall \alpha . \sigma \end{array}$$

Free type variables

$$\begin{array}{rcl} \mathsf{TV}(b) & = & \emptyset \\ \mathsf{TV}(\alpha) & = & \{\alpha\} \\ \mathsf{TV}(\sigma \to \tau) & = & \mathsf{TV}(\sigma) \cup \mathsf{TV}(\tau) \\ \mathsf{TV}(\forall \alpha . \sigma) & = & \mathsf{TV}(\sigma) - \{\alpha\} \end{array}$$

Free type variables in environments

$$\mathsf{TV}(\Gamma) \quad = \quad \bigcup_{x : \tau \in \Gamma} \mathsf{TV}(\tau)$$

λ_2 Derivation Rules

$$\frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathsf{TV}(\Gamma)}{\Gamma \vdash M : \forall \alpha . \sigma} \; (\forall\text{-}\mathit{Introduction})$$

$$\frac{\Gamma \vdash M : \forall \alpha . \sigma}{\Gamma \vdash M : \sigma[\alpha \mapsto \tau]} \; (\forall\text{-}\mathit{Elimination})$$

Does it solve our problem?

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_f \vdash f\,\mathbf{True} : \mathit{bool}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma_f \vdash f : \forall \alpha . \alpha \to \alpha}{\Gamma_f \vdash f : (\alpha \to \alpha)[\alpha \mapsto \mathit{int}]}}{\Gamma_f \vdash f : \mathit{int} \to \mathit{int}} \qquad \Gamma_f \vdash 5 : \mathit{int}}{\Gamma_f \vdash f\,5 : \mathit{int}} \qquad \Gamma_f \vdash 7 : \mathit{int}}{f : \forall \alpha . \alpha \to \alpha \vdash \mathbf{if}\ (f\,\mathbf{True})\ \mathbf{then}\ (f\,5)\ \mathbf{else}\ 7 : \mathit{int}}}{\vdash \lambda f . \mathbf{if}\ (f\,\mathbf{True})\ \mathbf{then}\ (f\,5)\ \mathbf{else}\ 7 : (\forall \alpha . \alpha \to \alpha) \to \mathit{int} \qquad (\vdash \lambda x . x : \forall \alpha . \alpha \to \alpha)}}{\vdash (\lambda f . \mathbf{if}\ (f\,\mathbf{True})\ \mathbf{then}\ (f\,5)\ \mathbf{else}\ 7)(\lambda x . x) : \mathit{int}}$$

Decidability

Type inference for $\lambda 2$ is undecidable

Let-polymorphism, a weak form of parametric polymorphism

- Quantifiers can occur only on the top-level of types
- Like this $\forall\alpha.(bool \to (\alpha \to \alpha) \to int)$
- But not $bool \to (\forall\alpha.\alpha \to \alpha) \to int$
- Type inference is decidable
- But less programs can be typed
- Haskell supports let-polymorphism

SL Now With Let-Polymorphism

Syntax

$$
\begin{array}{lcl}
e & ::= & x \\
 & | & \lambda x.e \\
 & | & e_1 e_2 \\
 & | & \textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e \\
 & | & e_1 \; op \; e_2 \\
 & | & i \mid \textbf{True} \mid \textbf{False} \\
 & | & \textbf{let } x = e_1 \textbf{ in } e_2 \\
 & | & (e_1, e_2) \mid \textbf{fst} \mid \textbf{snd} \\
 & | & [\,] \mid e_1 : e_2 \mid \textbf{null} \mid \textbf{head} \mid \textbf{tail} \\
op & ::= & + \mid \leq \mid \&\& \\
i & ::= & (0 \mid 1 \mid 2 \mid \ldots \mid 9)[i]
\end{array}
$$

Typing SL

Types

$$
\begin{array}{rcl}
\sigma & ::= & \alpha \\
& | & \sigma_1 \to \sigma_2 \\
& | & (\sigma_1, \sigma_2) \\
& | & [\sigma] \\
& | & \mathit{int} \mid \mathit{bool}
\end{array}
$$

Type Schemes

$$\Sigma ::= \forall \vec{\alpha}.\sigma$$

Environments

$\Gamma : \mathit{Variables} \to \Sigma$

Typing judgements:

$\Gamma \vdash E : \sigma$

Type Derivation Rules for SL (constants)

$$\frac{b \in \{\mathbf{True}, \mathbf{False}\}}{\Gamma \vdash b : \mathit{bool}}\ (\mathit{Bool}) \qquad \frac{i \in \{\ldots, -1, 0, 1, \ldots\}}{\Gamma \vdash i : \mathit{int}}\ (\mathit{Int})$$

$$\frac{\odot : \sigma_1 \to \sigma_2 \to \tau \qquad \Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash e_1 \odot e_2 : \tau}\ (\mathit{Bin\ op})$$

$$\frac{\Gamma \vdash e_c : \mathit{bool} \qquad \Gamma \vdash e_t : \sigma \qquad \Gamma \vdash e_e : \sigma}{\Gamma \vdash \mathbf{if}\ e_c\ \mathbf{then}\ e_t\ \mathbf{else}\ e_e : \sigma}\ (\mathit{If})$$

Type Derivation Rules (tuples)

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2) : (\sigma_1, \sigma_2)} \; \color{red}{(Tuple)}$$

$$\frac{}{\Gamma \vdash \mathbf{fst} : (\sigma_1, \sigma_2) \to \sigma_1} \; \color{red}{(Fst)}$$

$$\frac{}{\Gamma \vdash \mathbf{snd} : (\sigma_1, \sigma_2) \to \sigma_2} \; \color{red}{(Snd)}$$

Type Derivation Rules for SL (lists)

$$\frac{}{\Gamma \vdash [\,] : [\sigma]} \; (\mathit{Nil})$$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : [\sigma]}{\Gamma \vdash (e_1 : e_2) : [\sigma]} \; (\mathit{Cons})$$

$$\frac{}{\Gamma \vdash \mathbf{null} : [\sigma] \to \mathit{bool}} \; (\mathit{Null})$$

$$\frac{}{\Gamma \vdash \mathbf{head} : [\sigma] \to \sigma} \; (\mathit{Head})$$

$$\frac{}{\Gamma \vdash \mathbf{tail} : [\sigma] \to [\sigma]} \; (\mathit{Tail})$$

Type Derivation Rules for SL (functions, let)

$$\frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \; (App)$$

$$\frac{\Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \rightarrow \tau} \; (Abs)$$

$$\frac{\Gamma, x{:}\sigma \vdash e_1 : \sigma \qquad \Gamma, x{:}\forall\vec{\alpha}.\sigma \vdash e_2 : \tau \qquad \alpha_i \notin \mathsf{TV}(\Gamma)}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau} \; (Let)$$

$$\frac{x{:}\forall\vec{\alpha}.\sigma \in \Gamma}{\Gamma \vdash x : \sigma[\alpha_i \mapsto \tau_i]} \; (Var)$$

Examples

This expression still cannot be typed

```
(\f.if (f True) then (f 5) else 7)(\x.x)
```

But this one can be typed

```
let f = \x.x in if (f True) then (f 5) else 7
```

Try it in Haskell

Try to make the type derivation

Bibliography

Henk Barendregt, Erik Barendsen. “Introduction to Lambda Calculus”. 2000

Benjamin Pierce. “Types and Programming Languages”. MIT Press, 2002

Coming Up Next

Present your parser

Algorithm for polymorphic type inference