Introduction to Type Theory

Lecture 7

Adapted in part from Luca Cardelli’s tutorial and slides by George Necula.
Motivation

Consider the following program

\[ x := 5; \text{while } x \text{ do } x := x - 1; \]

What is the semantics of this program?

\[ \langle x := 5; \text{while } x \text{ then } x := x - 1, \sigma \rangle \rightarrow \sigma' \]

We are stuck!!

None of our derivation rules apply.
What to do in this situation?

Options

1) Leave it up to the implementation
   • that’s the C approach
   • is it a good idea?

2) Provide a mechanism to identify and rule out such “bad” programs
   • Programs can only run if you can prove they will execute to completion according to the semantics of the language
   • type systems will allow us to do this!

3) Prescribe correct behavior for every program
   • untyped λ-calculus works like this
   • do any practical languages do this?
   • Type systems are useful in this situation too.
What is a type system

Narrow View
- It’s a mechanism for ensuring that variables only take values from predefined sets
  • Ex. Integers, Strings, Characters
- A mechanism for avoiding unchecked errors
  • by ruling out programs with undefined behaviors
  • by specifying how a program should fail (eg. NullPointerException)

Expansive View
- It’s a light-weight proof system and annotation mechanism for efficiently checking for a specific property of interest
- Address bugs that go beyond corner-cases in the semantics
  • Information flow violations
  • deadlocks
  • atomicity violations
  • etc, etc, etc
Goals for this unit

Learn how to formally define a type system

Learn how to make arguments about the soundness of a type system

Learn about type checking and type inference

Understand the possibilities and limitations of type-based analyses
Formalizing a type system

The type system is almost never orthogonal to the semantics of the language

- The types in a program can affect its behavior (e.g. operator overloading)

We don’t define the type system in isolation, we define a typed *language* including definitions of

- The syntax
- dynamic semantics (e.g. operational semantics)
- static semantics
  - also known as typing rules
  - describe how types are assigned to elements in a program
- type soundness argument
  - describe the relationship between static and dynamic semantics
Basic notation

The type system assigns types to elements in the language

- basic notation: \( e : T \) (\( e \) is of type \( T \))
- What is the type of:
  
  5

The types of some elements depends on the environment

- basic notation \( \Gamma \vdash e : T \)
  
  (Given environment \( \Gamma \), we can derive that \( e \) is of type \( T \))
- An environment associates types with free variables
- This is called a Judgment
- Ex.
  
  \( x : \text{int}, y : \text{int} \vdash x + y : \text{int} \)
We know how to define the operational semantics of a language.

Types give us some additional flexibility.

- **Ex. Operator Overloading**

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow n_1 \quad \langle e_2, \sigma \rangle &\rightarrow n_2 \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow n
\end{align*}
\]
Static Semantics

Typing rules
- Typing rules tell us how to derive typing judgments
- Very similar to derivation rules in Big Step OS

\[
\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad \frac{\Gamma \vdash N : \text{int}}{\Gamma \vdash \text{e1} + \text{e2} : \text{int}}
\]

Ex. Language of Expressions

\[
\frac{\Gamma \vdash \text{e1} : \text{int} \quad \Gamma \vdash \text{e2} : \text{int}}{\Gamma \vdash \text{e1} + \text{e2} : \text{int}}
\]
Ex. Language of Expressions

\[ \frac{x : T \in \Gamma \quad \Gamma \vdash x : T}{\Gamma \vdash N : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \]

Show that the following Judgment is valid

\[ x : \text{int}, y : \text{int} \vdash x + (y + 5) : \text{int} \]

\[ \frac{x : \text{int}, y : \text{int} \vdash x : \text{int} \quad x : \text{int}, y : \text{int} \vdash (y + 5) : \text{int}}{x : \text{int}, y : \text{int} \vdash x + (y + 5) : \text{int}} \]

\[ \frac{x : \text{int} \in x : \text{int}, y : \text{int} \quad x : \text{int}, y : \text{int} \vdash y : \text{int} \quad x : \text{int}, y : \text{int} \vdash 5 : \text{int}}{x : \text{int}, y : \text{int} \vdash x + (y + 5) : \text{int}} \]
Simply Typed $\lambda$ Calculus ($F_1$)

Basic Typing Rules

\[
\begin{align*}
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} & \quad \frac{\Gamma, x : \tau \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau \ e) : \tau_1 \rightarrow \tau_2} & \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}
\end{align*}
\]

Extensions

\[
\begin{align*}
\frac{\Gamma \vdash N : \text{int}}{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}} & \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} & \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 = e_2 : \text{bool}}
\end{align*}
\]

\[
\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \text{if } e \text{ then } e_t \text{ else } e_f : \tau}
\]
Example

Is this a valid typing judgment?

\[ \vdash (\lambda x: \text{bool} \ \lambda y: \text{int} \ if \ x \ then \ y \ else \ y + 1): \text{bool} \to \text{int} \to \text{int} \]

How about this one?

\[ \vdash (\lambda x: \text{int} \ \lambda y: \text{bool} \ x + y): \text{int} \to \text{bool} \to \text{int} \]
Example

What's the type of this function?

\((\lambda f. \lambda x. \text{if } x = 1 \text{ then } x \text{ else } (f \ f \ (x-1)) \ast x)\)

\[\begin{align*}
x: \tau & \in \Gamma \\
\Gamma \vdash x : \tau
\end{align*}\]

\[\begin{align*}
\Gamma, x: \tau_1 \vdash e: \tau_2 \\
\Gamma \vdash (\lambda x: \tau_1 e): \tau_1 \to \tau_2
\end{align*}\]

\[\begin{align*}
\Gamma \vdash e_1: \tau' \to \tau \\
\Gamma \vdash e_2: \tau' \\
\Gamma \vdash e_1 e_2: \tau
\end{align*}\]

\[\begin{align*}
\Gamma \vdash e_1: \text{int} \\
\Gamma \vdash e_2: \text{int} \\
\Gamma \vdash e_1 + e_2: \text{int}
\end{align*}\]

\[\begin{align*}
\Gamma \vdash e_1: \text{int} \\
\Gamma \vdash e_2: \text{int} \\
\Gamma \vdash e_1 = e_2: \text{bool}
\end{align*}\]

\[\begin{align*}
\Gamma \vdash e: \text{bool} \\
\Gamma \vdash e_t : \tau \\
\Gamma \vdash e_f : \tau \\
\Gamma \vdash \text{if } e \text{ then } e_t \text{ else } e_f : \tau
\end{align*}\]

- Hint: This IS a trick question
We have defined a really strong type system on calculus. It's so strong, it won't even let us write non-terminating functions. We can actually prove this! In order to write non-terminating functions, we need to extend our type system with recursive types. We'll talk about those next week.

Simply Typed $\lambda$ Calculus ($\text{F}^1$)
Type Soundness Arguments

We want to establish a relationship between
- the static semantics (typing rules) and
- the dynamic (operational) semantics of the program

Key idea:
- Define what it means for a value to have a type \( v : \tau \)
- Define what it means for an expression to have a type \( \vdash e : \tau \)
  - The typing rules do this for us
- Show that \( \Gamma \vdash e : \tau \land (\langle e, \sigma \rangle \rightarrow v) \Rightarrow v : \tau \)
Example: Language of Expressions

- What does it mean for \( v \) to be \( \text{int}, \text{bool} \)

\[ v : \text{bool} \iff v \in \|\text{bool}\| = \{\text{true}, \text{false}\} \quad v : \text{int} \iff v \in \|\text{int}\| = \mathbb{N} \]

- How do we show that \( \forall \Gamma, \sigma, \tau. \Gamma \vdash e : \tau \land (\langle e, \sigma \rangle \to v) \Rightarrow v : \tau \) ?
  
  • Is this even true?

- Let’s try proving it by structural induction
Structural Induction

\[ P(e) := \forall_{\Gamma,\sigma,\tau} \quad \Gamma \vdash e : \tau \land \langle e, \sigma \rangle \rightarrow v \Rightarrow v : \tau \]

Base case 1 \( e = N \):

- \( \Gamma \vdash N : \text{int} \)
- \( \langle N, \sigma \rangle \rightarrow n \)

\[ \checkmark \]

Base case 2 \( e = x \):

- \( x : T \in \Gamma \)
- \( \Gamma \vdash x : T \)
- \( \langle x, \sigma \rangle \rightarrow \sigma(x) \)

\[ \times \]

- Is \( P(x) \) true?
- NO!
- We need more conditions to ensure consistency of the environments

\[ \forall_{\Gamma,\sigma,\tau} \quad (\forall_x \ x : \tau \in \Gamma \iff \sigma(x) \in \|\tau\|) \land \quad \Gamma \vdash e : \tau \land \langle e, \sigma \rangle \rightarrow v \Rightarrow v : \tau \]
Type Soundness for F₁

Recall our Typing Rules

\[
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \ e) : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ e_2 : \tau}
\]

And our CBN Evaluation Rules

\[
\frac{\quad}{x \rightarrow x} \quad \frac{e_1 \rightarrow \lambda x. \ e'_1 \quad e'_1[\alpha(e_2)/x] \rightarrow e_3}{e_1 \ e_2 \rightarrow e_3} \quad \frac{\lambda x. \ e \rightarrow \lambda x. \ e}{\lambda x. \ e \rightarrow \lambda x. \ e}
\]

We want to show that

\[
\Gamma \vdash e : \tau \land (e \rightarrow v) \Rightarrow v : \tau
\]
Induction on the Structure of the Derivation

\[ P(d) := \Gamma \vdash e : \tau \land (d \models_R e \to v) \Rightarrow v : \tau \]

Base case:

- \( e = v \), so property is trivially satisfied

Inductive case

\[
\begin{align*}
   e_1 \to \lambda x. e'_1 & \quad e'_1[\alpha(e_2)/x] \to e_3 \\
   e_1 e_2 & \to e_3
\end{align*}
\]
Induction on the Structure of the Derivation

Given \( \Gamma \vdash e_1 : t_{e12} \) we want to show that \( \Gamma \vdash e_2 : t_{e12} \).

- By our typing rule, we have

\[
\Gamma \vdash e_1 : t' \rightarrow t_{e12} \quad \Gamma \vdash e_2 : t'
\]

- And by the IH, we have that

\[
\lambda x. e_1 : t' \rightarrow t_{e12}
\]

- Which again by the typing rule

\[
\Gamma \vdash (\lambda x. t') : t_{e12} \rightarrow t_{e12}
\]

- Now, we need to show that

\[
\Gamma, x : t' \vdash e_1' : (\alpha(e_2)/x) : t_{e12}
\]

- And from our IH

\[
\Gamma \vdash e_1' : (\alpha(e_2)/x) : t_{e12} \Rightarrow \Gamma \vdash e_3 : t_{e12}
\]
So what did we just prove?

We prove that if $e: \tau$ evaluates to $v$, then $v: \tau$
- This is a pretty strong guarantee, but...
- This tells us nothing about termination

Turns out that if $\vdash e: \tau$ then $e$ is guaranteed to terminate!
- But that needs to be proved separately.