EDAP15: Program Analysis

MONOMORPHIC TYPE ANALYSIS

Christoph Reichenbach

Announcements

- Wednesdays: room $\rightarrow$ E:3308
- Mondays: room change requires time change; viable slots?
- Exercise 0 available after class

Types

Java

```
int v;
```

Haskell

```
v :: Int
```

ML

```
val v : int
```

- Framework for classifying parts of programs by:
  - Which set they may be drawn from, and/or
  - What behaviour they exhibit
- *Type analysis* deals with:
  - *Checking types*: Do the types agree?
  - *Inferring types*: Given part of a program, what is its type?
- We focus on *static type analysis*

Types and Programs: Two Languages

Language $\mathcal{V}$:

$$\begin{array}{rcl} \mathit{val} & ::= & \mathit{nat} \\ & | & \mathsf{true} \;|\; \mathsf{false} \end{array}$$

Language $\mathbb{T}_{\mathcal{V}}$:

$$\begin{array}{rcl} \mathit{type} & ::= & \textsc{Int} \\ & | & \textsc{Bool} \end{array}$$

- For program analysis, best to consider types and programs *separate* languages
  - Target language's type system may not match our needs
  - Language $\mathcal{V}$ entirely lacks type system
- Abstract over $\mathcal{V}$ with $\mathbb{T}_{\mathcal{V}}$:

$$\mathit{23} : \textsc{Int} \qquad\qquad \mathsf{true} : \textsc{Bool}$$

- From that perspective, "has-type-of" is a binary relation:

$$(:) \subseteq \mathcal{V} \times \mathbb{T}_{\mathcal{V}}$$

Uses of Type Analysis

- Types abstractly model program behaviour
- "Traditionally":
  - Set of possible computational results
  - Set of possible behaviours of computational result
- We can model other behaviour as types:
  - Uncaught exceptions
  - Use of shared memory regions
  - Other side effects
  - Dependencies
  - Race conditions in concurrent memory access

  ...

Applying Type Systems

Given program p: analyse $p : \tau$

Type Checking

- Assume τ is *given*
- Test: **Is $p : \tau$ true?**
- Can *use* type inference

Type Inference

- Assume τ is *not given*
- Find all τ s.th. $p : \tau$
  - None/Multiple: *Type Error*

Program Analysis Designer's View

- *Checking* τ requires specification
- Examples:
  - User spec: "no exceptions"
  - Language spec: "no side effects allowed here"

- *Inferring* τ can sensibly yield multiple results
  - Zero/many properties of interest
  - Example: τ describes type of exception that might be raised

Summary

- Types abstractly *model* some aspect of a program
- For a given analysis, the language of *types* and *programs* might be distinct
- Type analysis examines:
  - **Type Checking** Does this program have some specific type?
  - **Type Inference** Which types can this program have?
- Standard notation: the binary **typing relation** (:) relates programs p and their types τ:

$$p : \tau$$

A Simple Language: IGA

$$
\begin{array}{lcl}
\textit{expr} & ::= & \langle \textit{val} \rangle \\
 & | & \langle \textit{expr} \rangle \ \text{plus} \ \langle \textit{expr} \rangle \\
 & | & \langle \textit{expr} \rangle >= \langle \textit{expr} \rangle \\
 & | & \text{if} \ \langle \textit{expr} \rangle \ \text{then} \ \langle \textit{expr} \rangle \ \text{else} \ \langle \textit{expr} \rangle \\
 & & \\
\textit{val} & ::= & \textit{nat} \\
 & | & \text{true} \ | \ \text{false} \\
 & & \\
\textit{nat} & ::= & 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ \ldots
\end{array}
$$

- Semantics mostly straightforward:
- plus operates only on *nat*
- $>=$ requires *nat* arguments and returns true or false
- **if** e_1 **then** e_2 **else** e_3:
  - If e_1 evaluates to true: computes e_2
  - If e_1 evaluates to false: computes e_3

The Typing Relation

- We the set of types of IGA, $\mathbb{T}_{iga} = \{\textsc{Bool}, \textsc{Int}\}$:
  - $\textsc{Bool}$: Type of booleans (true, false)
  - $\textsc{Int}$: Type of natural numbers (0, 1, 2, ...)
- We can now type values:

$$\begin{array}{lcl} \mathsf{true} & : & \textsc{Bool} \\ 23 & : & \textsc{Int} \end{array}$$

- Correspondibly (:) is a binary relation:

$$(:) \subseteq val \times \mathbb{T}_{iga}$$

Types for Values

- To analyse all of IGA, we extend (:) to expressions:

$$(:) \subseteq \mathit{expr} \times \mathbb{T}_{iga}$$

- We want to type e.g.:

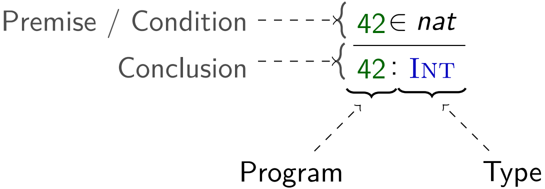
$$\text{39 plus 3} \quad : \quad \textsc{Int}$$

For clarity, we will write this formally

Types for Expressions

$$\frac{}{\mathsf{true} : \textsc{Bool}}\ (\textit{t-true}) \qquad \frac{}{\mathsf{false} : \textsc{Bool}}\ (\textit{t-false}) \qquad \frac{v \in nat}{v : \textsc{Int}}\ (\textit{t-nat})$$

Conditional Typing Rules



If $42 \in nat$ holds, then so does $42 : \textsc{Int}$

- v is a *Metavariable*
- We can replace v by *anything*
  - One restriction: we must do so *everywhere in the rule at once*

$\Rightarrow$ "*Substitution*"

Types for Expressions

$$\frac{}{\text{true} : \textsc{Bool}}\ (\textit{t-true}) \qquad \frac{}{\text{false} : \textsc{Bool}}\ (\textit{t-false}) \qquad \frac{v \in \mathit{nat}}{v : \textsc{Int}}\ (\textit{t-nat})$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1\ \text{plus}\ e_2 : \textsc{Int}}\ (\textit{t-plus})$$

Recursive Typing Rules

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \ (\textit{t-plus}) \qquad \frac{v \in \textit{nat}}{v : \textsc{Int}} \ (\textit{t-nat})$$

$$\boxed{\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \ (\textit{t-plus})} \begin{bmatrix} e_1 & \mapsto & 1 \\ e_2 & \mapsto & 2 \text{ plus } 3 \end{bmatrix}$$

$$1 \text{ plus } 2 \text{ plus } 3 : \textsc{Int}$$

Recursive Typing Rules

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \; (\textit{t-plus})$$

$$\frac{v \in \mathit{nat}}{v : \textsc{Int}} \; (\textit{t-nat})$$

$$\frac{1 : \textsc{Int} \qquad 2 \text{ plus } 3 : \textsc{Int}}{1 \text{ plus } 2 \text{ plus } 3 : \textsc{Int}} \; (\textit{t-plus})$$

Recursive Typing Rules

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \ (\textit{t-plus})$$

$$\frac{v \in \textit{nat}}{v : \textsc{Int}} \ (\textit{t-nat})$$

$$\boxed{\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \ (\textit{t-plus})} \begin{bmatrix} e_1 & \mapsto & 2 \\ e_2 & \mapsto & 3 \end{bmatrix}$$

$$\frac{1 : \textsc{Int} \qquad\qquad 2 \text{ plus } 3 : \textsc{Int}}{1 \text{ plus } 2 \text{ plus } 3 : \textsc{Int}} \ (\textit{t-plus})$$

Recursive Typing Rules

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \; (\textit{t-plus}) \qquad \frac{v \in nat}{v : \textsc{Int}} \; (\textit{t-nat})$$

$$\frac{1 : \textsc{Int} \qquad \dfrac{2 : \textsc{Int} \qquad 3 : \textsc{Int}}{2 \text{ plus } 3 : \textsc{Int}} \; (\textit{t-plus})}{1 \text{ plus } 2 \text{ plus } 3 : \textsc{Int}} \; (\textit{t-plus})$$

Recursive Typing Rules

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \; (\textit{t-plus}) \qquad \frac{v \in \textit{nat}}{v : \textsc{Int}} \; (\textit{t-nat})$$

$[v \mapsto 1]$ $[v \mapsto 2]$ $[v \mapsto 3]$

$$\frac{1 \in \textit{nat}}{1 : \textsc{Int}} \; (\textit{t-nat}) \qquad \frac{2 \in \textit{nat}}{2 : \textsc{Int}} \; (\textit{t-nat}) \qquad \frac{3 \in \textit{nat}}{3 : \textsc{Int}} \; (\textit{t-nat})$$

$$\frac{1 : \textsc{Int} \qquad \dfrac{2 : \textsc{Int} \qquad 3 : \textsc{Int}}{2 \text{ plus } 3 : \textsc{Int}} \; (\textit{t-plus})}{1 \text{ plus } 2 \text{ plus } 3 : \textsc{Int}} \; (\textit{t-plus})$$

Recursive Typing Rules

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}}\ (\textit{t-plus}) \qquad\qquad \frac{v \in \textit{nat}}{v : \textsc{Int}}\ (\textit{t-nat})$$

$$\frac{\dfrac{1 \in \textit{nat}}{1 : \textsc{Int}}\ (\textit{t-nat}) \qquad \dfrac{\dfrac{2 \in \textit{nat}}{2 : \textsc{Int}}\ (\textit{t-nat}) \quad \dfrac{3 \in \textit{nat}}{3 : \textsc{Int}}\ (\textit{t-nat})}{2 \text{ plus } 3 : \textsc{Int}}\ (\textit{t-plus})}{1 \text{ plus } 2 \text{ plus } 3 : \textsc{Int}}\ (\textit{t-plus})$$

Types for Expressions

$$\frac{}{\mathsf{true} : \textsc{Bool}}\ (\textit{t-true}) \qquad \frac{}{\mathsf{false} : \textsc{Bool}}\ (\textit{t-false}) \qquad \frac{v \in \mathit{nat}}{v : \textsc{Int}}\ (\textit{t-nat})$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \ \mathsf{plus}\ e_2 : \textsc{Int}}\ (\textit{t-plus}) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}}\ (\textit{t-ge})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ (\textit{t-if})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \textsc{Int} \quad e_3 : \textsc{Int}}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \textsc{Int}}\ (\textit{t-if-nat})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \textsc{Bool} \quad e_3 : \textsc{Bool}}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \textsc{Bool}}\ (\textit{t-if-bool})$$

(*if*) rule summarises (*if-nat*) and (*if-bool*) via *metavariable*

Checking Types

- With $e : \tau$, we can have:

1. Exactly one τ fits (we've computed a type):

$$2 \text{ plus } 3 : \textsc{Int}$$

2. No τ fits (type error):

Type error in true plus 0

3. Multiple τ fit: can't happen in this type system

Inferring Types

- Checking explores "*is everything consistent?*"
- Inferring explores "*what is possible?*"
- In program analysis, we often want the latter. Recall:

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau} \ (\textit{t-if})$$

- What if we don't care for consistency and instead simply want to know all options (e.g., for optimisation)?
- The following rule may be a better fit:

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau_2 \quad e_3 : \tau_3 \quad i \in \{2, 3\}}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau_i} \ (\textit{t-if}')$$

- For efficiency, can store types in sets ('Set Types'):

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau_2 \quad e_3 : \tau_3}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau_2 \cup \tau_3} \ (\textit{t-if}'')$$

- Can design type rules so set types always produce one type.

Summary

- *Type systems* relate expressions to types:

$$(:) \subseteq expr \times \mathbb{T}_{iga}$$

- We use *inference rules* to compactly describe the type system

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau} \; (\textit{t-if})$$

- No type matches $\Rightarrow$ type error
- We will focus on Type Checking for a bit, for simplicity

Adding Variables: The language INGA

$$
\begin{array}{lcll}
\mathit{expr} & ::= & \langle \mathit{val} \rangle & \\
 & | & \mathit{id} & \textbf{new!} \\
 & | & \text{let } \mathit{id} = \langle \mathit{expr} \rangle \text{ in } \langle \mathit{expr} \rangle & \textbf{new!} \\
 & | & \langle \mathit{expr} \rangle \text{ plus } \langle \mathit{expr} \rangle & \\
 & | & \langle \mathit{expr} \rangle >= \langle \mathit{expr} \rangle & \\
 & | & \text{if } \langle \mathit{expr} \rangle \text{ then } \langle \mathit{expr} \rangle \text{ else } \langle \mathit{expr} \rangle & \\
 & & & \\
\mathit{val} & ::= & \mathit{nat} & \\
 & | & \text{true} \mid \text{false} & \\
 & & & \\
\mathit{nat} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \ldots & \\
\mathit{id} & ::= & \underline{x} \mid \underline{y} \mid \underline{z} \mid \ldots &
\end{array}
$$

- Adds locally scoped variable bindings
- let $\underline{x}$ = 1 plus 2 in $\underline{x}$ + 3 evaluates to 6
- let $\underline{x}$ = 1 in (let $\underline{x}$ = 2 in $\underline{x}$) + $\underline{x}$ evaluates to 3

Typing Variables

$$\frac{}{\text{true} : \textsc{Bool}}\ (\textit{t-true}) \qquad \frac{}{\text{false} : \textsc{Bool}}\ (\textit{t-false}) \qquad \frac{v \in nat}{v : \textsc{Int}}\ (\textit{t-nat})$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}}\ (\textit{t-plus}) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}}\ (\textit{t-ge}) \qquad \frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ (\textit{t-if})$$

- Same types as before: $\mathbb{T}_{inga} = \{\textsc{Bool}, \textsc{Int}\}$
- Need new typing rules for let and variables:

$$\frac{}{\underline{x} : \tau}\ \textit{t-var}$$

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2}\ \textit{t-let}$$

How do we connect τ_1 and τ and τ_2 ?

Connecting Variables and Types

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2} \textit{ t-let} \quad \overset{?}{\longleftrightarrow} \quad \frac{}{\underline{x} : \tau} \textit{ t-var}$$

- We know that $\underline{x} : \tau_1$ before we analyse e_2
- Must carry this information into the analysis of e_2:
- Can be solved with typing rules with a bit of extra notation:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[\underline{x} \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2} \textit{ t-let}$$

This notation does't reflect how we would solve name/type analysis in Java / Scala / JastAdd.

Variables and Types in Practice

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2}\ \textit{t-let} \quad \overset{?}{\longleftrightarrow} \quad \frac{}{\underline{x} : \tau}\ \textit{t-var}$$

- Instead, we will "cheat" (*or, rather, use a notational trick*):
  - Write $\boxed{\underline{x}.\mathtt{ty} = \tau}$ to *assert* type of $\underline{x}$
  - Semantics:
    - If $\underline{x}$`.ty` unset, assign τ
    - Otherwise check equality
  - For now only works if we analyse top-down (more later, though!)
- **Note**: these attributes are associated with the *variable declarations* / *symbol table entries*:

$$\text{let } \underline{x} = 1 \text{ in let } \underline{x} = \text{true in } \underline{x} : \textsc{Bool}$$

- Equivalently, assume that all variables have unique names.

Typing INGA

$$\frac{}{\text{true} : \textsc{Bool}} \ (\textit{t-true}) \qquad \frac{}{\text{false} : \textsc{Bool}} \ (\textit{t-false}) \qquad \frac{v \in nat}{v : \textsc{Int}} \ (\textit{t-nat})$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}} \ (\textit{t-plus}) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}} \ (\textit{t-ge})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau} \ (\textit{t-if})$$

$$\frac{e_1 : \tau_1 \quad \underline{x}.\mathtt{ty} = \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2} \ \textit{t-let} \qquad \frac{\underline{x}.\mathtt{ty} = \tau}{\underline{x} : \tau} \ \textit{t-var}$$

Example

$$\frac{}{\text{true} : \textsc{Bool}}\ (\textit{t-true}) \qquad \frac{}{\text{false} : \textsc{Bool}}\ (\textit{t-false}) \qquad \frac{v \in nat}{v : \textsc{Int}}\ (\textit{t-nat})$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}}\ (\textit{t-plus}) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}}\ (\textit{t-ge}) \qquad \frac{\underline{x}.\texttt{ty} = \tau}{\underline{x} : \tau}\ (\textit{t-var})$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ (\textit{t-if}) \qquad \frac{e_1 : \tau_1 \quad \underline{x}.\texttt{ty} = \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2}\ (\textit{t-let})$$

$$\boxed{\underline{x}.\texttt{ty} = \textsc{Int}}$$

$$\frac{\dfrac{1 \in nat}{1 : \textsc{Int}}\ (\textit{t-nat}) \quad \underline{x}.\texttt{ty} = \textsc{Int} \quad \dfrac{\dfrac{\underline{x}.\texttt{ty} = \textsc{Int}}{\underline{x} : \textsc{Int}}\ (\textit{t-var}) \quad \dfrac{\underline{x}.\texttt{ty} = \textsc{Int}}{\underline{x} : \textsc{Int}}\ (\textit{t-var})}{\underline{x} \text{ plus } \underline{x} : \textsc{Int}}\ (\textit{t-plus})}{\text{let } \underline{x} = 1 \text{ in } \underline{x} \text{ plus } \underline{x} : \textsc{Int}}\ (\textit{t-let})$$

Summary

- To analyse realistic programs, we must analyse name bindings
- We do so through *indirection*:
  - Assume that we associate names with declarations / symbol table entries
  - When we encounter a name and want to:
    - *type-check*: if type not bound, set it now, otherwise check
    - *read type*: only works if type is already bound

Outlook

- **Remember**:
  - Labs for Exercise 0 tomorrow and Friday, 08:00 E:Gamma
  - Check for Videos and Quizzes tomorrow
  - Form groups by today, 18:00!
- Next Lecture: Monday
  - Polymorphic Type Analysis

`http://cs.lth.se/EDAP15`