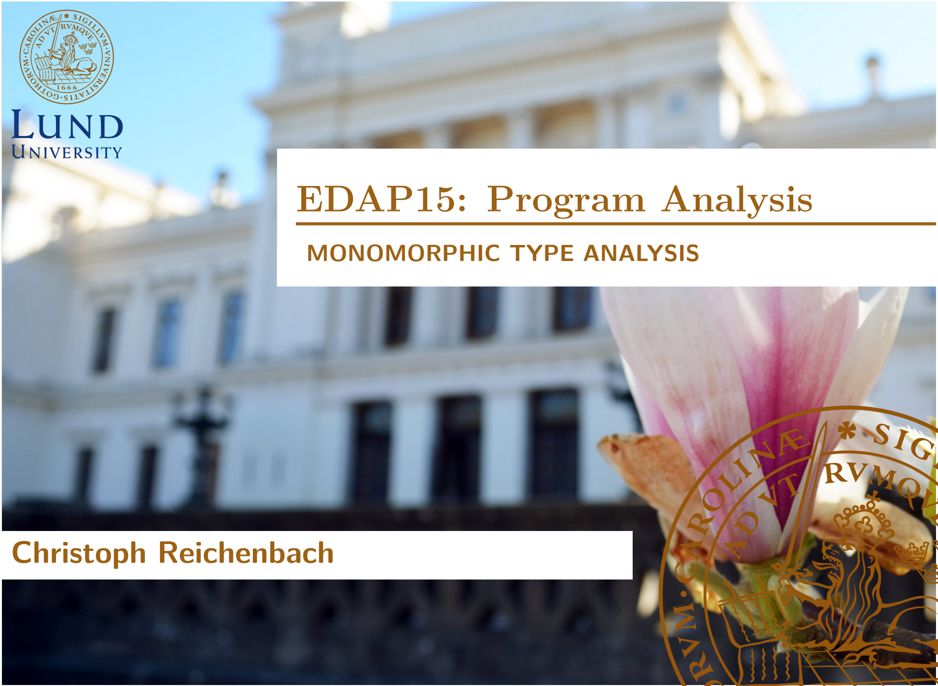


EDAP15: Program Analysis

MONOMORPHIC TYPE ANALYSIS

Christoph Reichenbach
Lund University

Lecture Overview

Foundations	Static Analysis: Properties	Static Analysis: Control Flow	Dynamic Analysis
01 Foundations	03 04 **Types**		12 Instrumentation
02 Constructing Program Analyses in JastAdd	05 06 07 Data Flow	05 Intraprocedural	13 Analysis
	08 09 Memory	10 Interprocedural	
		11 Indirect	

14 Review

Announcements

- Quizzes for Lecture 2 were hidden by accident

$\Longrightarrow$ now optional

  - Still strongly recommended, especially if you have not taken EDAN65

Notation Warning

JastAdd BNF Grammar

```
// start symbol:
Program ::= Expr;

abstract Expr;
IntConstant :  Expr ::= <Val:int>;
NullConstant :  Expr;
AddExpr :  Expr ::= L:Expr R:Expr;
SubExpr :  Expr ::= L:Expr R:Expr;
```

Compact BNF Grammar

$$
\begin{array}{lcl}
\mathit{Program} & ::= & \langle \mathit{Expr} \rangle \\
\\
\mathit{Expr} & ::= & 0 \mid 1 \mid \text{-}1 \mid \ldots \\
& \mid & \text{null} \\
& \mid & \langle \mathit{Expr} \rangle + \langle \mathit{Expr} \rangle \\
& \mid & \langle \mathit{Expr} \rangle \text{ - } \langle \mathit{Expr} \rangle
\end{array}
$$

- For brevity, we also use the compact BNF notation above
  - Equivalent to JastAdd Abstract Grammar notation
  - No production names:
    - Less verbose
    - Use terminal symbols to distinguish production rules

Types

Java

```
int v;
```

Haskell

```
v :: Int
```

ML

```
val v : int
```

- Types describe:
  - Set of posssible results
  - Possible *side effects*
    (e.g., Java's `f() throws IOException`, Haskell's **IO** monad)
- *Type analysis* deals with:
  - *Checking*: Do the types agree?
  - *Inference*: What types are possible?
- We focus on *static type analysis*

Types and Programs: Two Languages

Language $\mathcal{V}$:

$$
\begin{array}{lcl}
\mathit{Val} & ::= & \mathsf{true} \\
 & \mid & \mathsf{false} \\
 & \mid & \langle \mathit{Nat} \rangle
\end{array}
$$

$$
\mathit{Nat} \quad ::= \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \ldots
$$

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

$$
\begin{array}{lcl}
\mathit{Type} & ::= & \textsc{Bool} \\
 & \mid & \textsc{Int}
\end{array}
$$

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

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

- "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 permitted inputs
  - Set of possible outputs
- Advanced type analyses can model *side effects*:
  - File or network access
  - Exceptions
  - Software tools also use type analysis to find:
  - Dependencies
  - Use of shared memory regions
  - Race conditions in concurrent memory access

  ...

Two Types of Applications

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. $p : \tau$

Program Analysis Designer's View

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

- *Inferring* τ may yield 0, 1, or more results
  - Some analyses allow this
  - 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* are usually distinct
- Type analysis examines:
  - **Type Inference** Which types can this program have?
  - **Type Checking** Does this program have some specific type?
    - Often *uses* type inference
- Standard notation: the binary **typing relation** (:) relates programs p and their types τ:

$$p : \tau$$

A Simple Language: IGA

$$
\begin{array}{lcl}
\mathit{Expr} & ::= & \langle \mathit{Val} \rangle \\
 & | & \langle \mathit{Expr} \rangle \ \text{plus} \ \langle \mathit{Expr} \rangle \\
 & | & \langle \mathit{Expr} \rangle >= \langle \mathit{Expr} \rangle \\
 & | & \text{if} \ \langle \mathit{Expr} \rangle \ \text{then} \ \langle \mathit{Expr} \rangle \ \text{else} \ \langle \mathit{Expr} \rangle \\
 & & \\
\mathit{Val} & ::= & \langle \mathit{Nat} \rangle \\
 & | & \text{true} \ | \ \text{false} \\
 & & \\
\mathit{Nat} & ::= & 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ \ldots
\end{array}
$$

- 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:

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

- Thus:

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

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

Premise / Condition

Conclusion

$$\frac{42 \in \mathit{Nat}}{\underbrace{42}_{\text{Program}} : \underbrace{\textsc{Int}}_{\text{Type}}}$$

If $42 \in \mathit{Nat}$ 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

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

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

Recursive Typing Rules

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

$$\Downarrow$$

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

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

Recursive Typing Rules

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

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

Recursive Typing Rules

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

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

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

Recursive Typing Rules

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

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

Recursive Typing Rules

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

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

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

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

Recursive Typing Rules

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

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

Types for Expressions

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

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

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

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

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

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

Checking Types

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

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

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

2. No τ fits (type error):

$$\text{true plus 0 } \textit{has type error}$$

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:

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

Summary

- *Type systems* relate expressions to types:

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

- 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 now

Question

How would we implement this kind of type analysis in JastAdd?

$$\frac{v \in Nat}{v : \textsc{Int}} \ (\textit{t-nat}) \qquad\qquad \frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau} \ (\textit{t-if})$$

JastAdd BNF Grammar

```
// start symbol:
Program ::= Expr;

abstract Expr;
IntConstant :  Expr ::= <Val:int>;
TrueConstant :  Expr;
FalseConstant :  Expr;
IfThenElseExpr :  Expr ::= Cond:Expr True:Expr False:Expr;
```

Adding Variables: The language INGA

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

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

Typing Variables

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

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

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

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

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

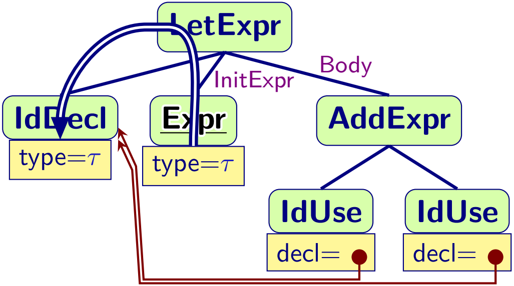
How do we connect τ_1 and τ and τ_2 ?

Connecting Variables and Types

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

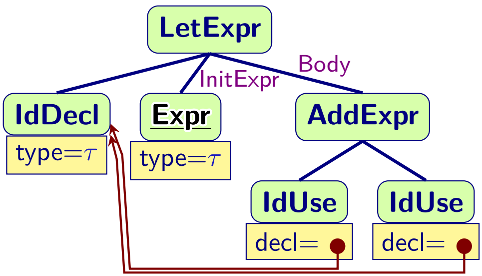
Example: let $\underline{x}$ = 1 in $\underline{x}$ + $\underline{x}$

- Using RAGs, we can compute $\underline{x} : \tau_1$ before analysing e_2:



1. **LetExpr** requests type of `InitExpr`
2. Passes type to **IdDecl** (via inherited attribute)
3. Any **IdUse** can now obtain type via `decl().type()`

Simple RAG Solution: Limitations



1. LetExpr: use `InitExpr.type()`
2. IdDecl: obtain `type()` from LetExpr via inherited attribute
3. IdUse: `decl().type()`

- Is this always well-defined?Limitations:
  We will only look at this case
  - **Name analysis must not depend on type analysis**
    - Always true for C, Pascal, Teal-0, ...
    - *Not always true* for Java, C++, ...
    - Requires *mutual recursion* (⇒ “circular attributes”, later...)
  - Expr's type must not depend on IdDecl's type
    - No recursive definitions allowed!
    - Not guaranteed for most modern languages
    - We want to support this!

Variables and Types, Formally

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{\mathsf{let}\ \underline{x} = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textit{t-let} \qquad \xleftrightarrow{\ ?\ } \qquad \frac{}{\underline{x} : \tau}\ \textit{t-var}$$

- Assume: name analysis does not depend on type analysis
  $\Longrightarrow$ Can identify each name by its declaration (`decl()`)
- Formalising:
  - Write $\boxed{\Delta(\underline{x}) = \tau}$ to *assert* type of $\underline{x}$
- Semantics:
  - Treat $\Delta(\underline{x})$ as attribute on $\underline{x}$
  - For each $\underline{x}$:
    - Must have at least one type assertion
    - Each $\Delta(\underline{x}) = \tau$ must assert same type τ

$\Rightarrow$ *Collection Attribute*, cf. prep video for next lecture

Typing INGA

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

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

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

$$\frac{e_1 : \tau_1 \quad \Delta(\underline{x}) = \tau_1 \quad e_2 : \tau_2}{\mathsf{let}\ \underline{x} = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textit{t-let} \qquad \frac{\Delta(\underline{x}) = \tau}{\underline{x} : \tau}\ \textit{t-var}$$

Example

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

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

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

$$\boxed{\Delta(\underline{x}) = \textsc{Int}}$$

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

Summary

- Type analysis of realistic programs requires name analysis
- Distinguish if name analysis $\leftrightarrow$ type analysis dependency and/or recursive definitions allowed:
  1. name $\not\leftrightarrow$ type analysis, forbid recursive definitions:
     $\Longrightarrow$ Straightforward (see earlier slides)
  2. name $\not\leftrightarrow$ type analysis, allow recursive definitions:
     $\Longrightarrow$ Requires *collecting constraints*, using Δ
  3. name $\leftrightarrow$ type analysis:
     $\Longrightarrow$ More complex approaches (not covered here)
- General approach: for each name, collect all type constraints
  - For now: assume constraints on same variable must be identical
- Other approaches:
  - Instead of $\Delta(\underline{x}) = \tau$: collect arbitrarily complex constraints
  - Instead of Δ, can use *environments* Γ
    - Original formalisation (Hindley/Damas/Milner)
    - More complex: must recursively "thread" Γ through typing rules
    - solves name analysis $\leftrightarrow$ type analysis dependency
    - supports some more advanced type analyses

Adding Lists: The Language LINGA

$$
\begin{array}{lcll}
\mathit{Expr} & ::= & \langle \mathit{Val} \rangle & \\
& | & \mathit{id} & \\
& | & \text{let } \mathit{id} = \langle \mathit{Expr} \rangle \text{ in } \langle \mathit{Expr} \rangle & \\
& | & \text{nil} & \textbf{new!} \\
& | & \text{cons } (\langle \mathit{Expr} \rangle, \langle \mathit{Expr} \rangle) & \textbf{new!} \\
& | & \langle \mathit{Expr} \rangle \text{ plus } \langle \mathit{Expr} \rangle & \\
& | & \langle \mathit{Expr} \rangle >= \langle \mathit{Expr} \rangle & \\
& | & \text{if } \langle \mathit{Expr} \rangle \text{ then } \langle \mathit{Expr} \rangle \text{ else } \langle \mathit{Expr} \rangle & \\
\\
\mathit{Val} & ::= & \mathit{nat} & \\
& | & \text{true} \ | \ \text{false} &
\end{array}
$$

- nil is the empty list
- cons(v, ℓ) takes list ℓ and prepends v
- Can express list [0, 1, 2] as:

$$\text{cons}(0, \text{cons}(1, \text{cons}(2, \text{nil})))$$

The Type of Lists

The language of types $\mathbb{T}_{linga}$ has one new production:

$$Ty ::= \textsc{Int} \mid \textsc{Bool} \mid \textsc{List}\,[\langle Ty \rangle]$$

Example types:

- cons(true, nil) : $\textsc{List}[\textsc{Bool}]$
- cons(1, cons(2, nil)) : $\textsc{List}[\textsc{Int}]$
- cons(1, cons(false, nil)) : *type error*
- cons(cons(1, nil), nil) : $\textsc{List}[\textsc{List}[\textsc{Int}]]$
- nil : $\textsc{List}[\textsc{Int}]$
  $\textsc{List}[\textsc{Bool}]$
  $\textsc{List}[\textsc{List}[\textsc{Int}]]$
  $\textsc{List}[\ldots, \textsc{List}[\textsc{Bool}], \ldots]$

First attempt at typing rules:

$$\frac{\tau \in \mathbb{T}_{linga}}{\text{nil} : \textsc{List}[\tau]}\ (\textit{t-nil})$$

$$\frac{e_1 : \tau \quad e_2 : \textsc{List}[\tau]}{\text{cons}(e_1, e_2) : \textsc{List}[\tau]}\ (\textit{t-cons})$$

nil has infinitely many of the types in $\mathbb{T}_{linga}$

Principal Types

- $\mathtt{p} : \tau$ may have infinitely many τ, can't process all
- General approach: Find **Principal Type**

$=$ One *single* type that subsumes all possible types

- May need to extend type language to express such principal types
- Various approaches:
  - *Parametric Polymorphism*, using parametric types
  - Subtype Polymorphism
  - Type Classes

  ...
- Here: use **Parametric Types** with **Type Variables**:
  - $\textsc{List}[\alpha]$ summarises $\textsc{List}[\textsc{Int}]$, $\textsc{List}[\textsc{Bool}]$, $\textsc{List}[\textsc{List}[\ldots]]$

Type Variables

$$
\begin{array}{lcl}
\textit{Ty} & ::= & \textsc{Int} \\
& | & \textsc{Bool} \\
& | & \textsc{List}\,[\langle \textit{Ty} \rangle] \\
& | & \textit{tyvar}
\end{array}
$$

$$\textit{tyvar} \quad ::= \quad \alpha \;|\; \beta \;|\; \gamma \;|\; \ldots$$

- Working with infinitely many types is impractical
- Summarise types by introducing **type variables** into $\mathbb{T}_{\textit{linga}}$
- Can now define **parametric type** of nil:

$$\frac{}{\mathsf{nil} : \textsc{List}[\alpha]}\ (\textit{t-nil})$$

- Still needs some tweaking, as we will see in the next lecture

Parametric Types can compactly summarise infinitely many types

Summary

- Precise analyses often need to know parameterise types with other types
  - $\textsc{List}[\textsc{List}[\textsc{Int}]]$
- Naïve *Recursive Types* are difficult to work with:
  - If we *don't* know a 'component type', we have potentially infinitely many types to remember
- **Parametric Types**:
  - $\textsc{List}[\alpha]$
  - Use **Type Variable** α to express that we know the type only *partially*
- **Principal Types**:
  - τ is *principal* for e if it *subsumes* all τ' with e : τ' (meaning of "subsumes" varies by type system/analysis)

Three Languages With Variables

Meta-Language

- Describes *Object Language*(s)
- Variables refer to object language concepts:
  - LINGA programs
  - $\mathbb{T}_{\mathit{linga}}$ types

Programs: LINGA

- "Object Language" #1
- Variables refer to input programs
- Example: <u>x</u> in

let <u>x</u> = 1 in <u>x</u>

Types: $\mathbb{T}_{\mathit{linga}}$

- "Object Language" #2
- Variables refer to unknown types
- Example: α in

$\textsc{List}[\alpha]$

Meta-Variables Can Reference Object-Language Variables

Meta-Variable references	Example	Meta-Variable Notation
Program	1 plus 2	e
Type	$\textsc{List}[\textsc{Bool}]$	τ
Program variable	<u>foo</u>	$\underline{x}$
Type Variable	α	α

Outlook

- Thursday: Polymorphic Type Analysis
  - Partly flipped lecture, make sure to prepare!
  - Videos up on Moodle
- Optionally available:
  - Refresher on formal notation
  - Review of "Meta-languages": languages for describing languages

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