



LUND  
UNIVERSITY

# EDAP15: Program Analysis

---

MONOMORPHIC TYPE ANALYSIS

Christoph Reichenbach



# Lecture Overview

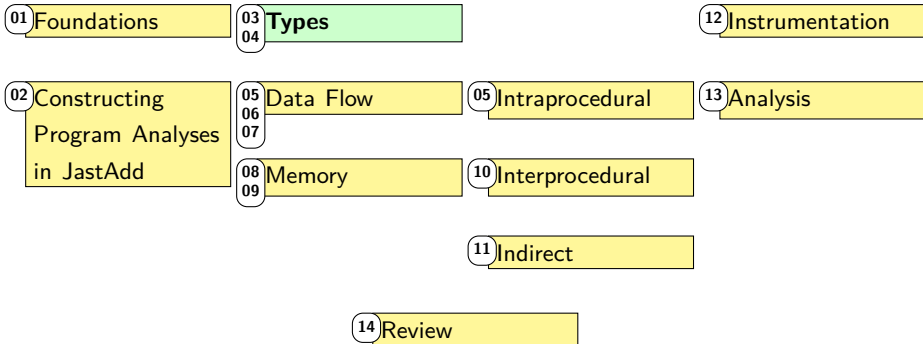
Foundations

Static Analysis

Dynamic  
Analysis

Properties

Control Flow



# Announcements

- ▶ Office hours: Thursdays 13:00–14:00 in E:2112B

# 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

*Program* ::=  $\langle Expr \rangle$

*Expr* ::=  $0 \mid 1 \mid -1 \mid \dots$   
| **null**  
|  $\langle Expr \rangle + \langle Expr \rangle$   
|  $\langle Expr \rangle - \langle Expr \rangle$

- ▶ 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 possible 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  $\mathcal{V}$ :

$Val ::= true$   
      |  $false$   
      |  $\langle Nat \rangle$

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

$Type ::= BOOL$   
      |  $INT$

$Nat ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \dots$

- ▶ 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  $\mathbb{T}_{\mathcal{V}}$ :

$23 : INT$

$true : BOOL$

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

## Type Inference

- ▶ Assume  $\tau$  is *not given*
- ▶ Find all  $\tau$  s.th.  $p : \tau$

## Program Analysis Designer's View

- ▶ *Checking*  $\tau$  requires specification
- ▶ *Inferring*  $\tau$  may yield 0, 1, or more results
  - ▶ Some analyses allow this
  - ▶ Example:  $\tau$  describes type of exception that might be raised
- ▶ Examples:
  - ▶ User spec: “no exceptions”
  - ▶ Language spec: “no side effects allowed here”

# 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  $\tau$ :

$p : \tau$

# A Simple Language: IGA

$$\begin{aligned} \text{Expr} & ::= \langle \text{Val} \rangle \\ & \quad | \langle \text{Expr} \rangle \text{ plus } \langle \text{Expr} \rangle \\ & \quad | \langle \text{Expr} \rangle \text{ } \geq \text{ } \langle \text{Expr} \rangle \\ & \quad | \text{if } \langle \text{Expr} \rangle \text{ then } \langle \text{Expr} \rangle \text{ else } \langle \text{Expr} \rangle \end{aligned}$$
$$\begin{aligned} \text{Val} & ::= \langle \text{Nat} \rangle \\ & \quad | \text{true} \quad | \quad \text{false} \end{aligned}$$
$$\text{Nat} ::= 0 \quad | \quad 1 \quad | \quad 2 \quad | \quad 3 \quad | \quad 4 \quad | \quad \dots$$

- ▶ 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,  $\mathbb{T}_{iga} = \{\text{BOOL}, \text{INT}\}$ :
  - ▶ **BOOL**: Type of booleans (**true**, **false**)
  - ▶ **INT**: Type of natural numbers (**0**, **1**, **2**, ...) )
- ▶ We can now type values:

**true** : **BOOL**  
**23** : **INT**

- ▶ Thus:

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

# Types for Values

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

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

- ▶ We want to type e.g.:

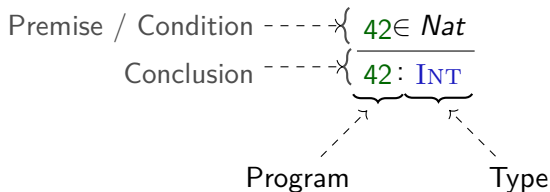
39 plus 3 : INT

**For clarity, we introduce concise, formal notation**

# Types for Expressions

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true})$$
$$\frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false})$$
$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

# Conditional Typing Rules



If  $42 \in \text{Nat}$  holds, then so does  $42 : \text{INT}$

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

# Types for Expressions

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true})$$

$$\frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

# Recursive Typing Rules

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$



$$\boxed{\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})} \left[ \begin{array}{l} e_1 \mapsto 1 \\ e_2 \mapsto 2 \text{ plus } 3 \end{array} \right]$$

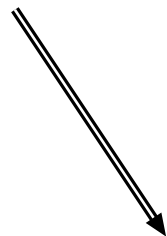
$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

1 plus 2 plus 3 : INT

# Recursive Typing Rules

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$



$$\frac{1 : \text{INT} \quad 2 \text{ plus } 3 : \text{INT}}{1 \text{ plus } 2 \text{ plus } 3 : \text{INT}} \quad (t\text{-plus})$$

# Recursive Typing Rules

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\boxed{\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})} \left[ \begin{array}{l} e_1 \mapsto 2 \\ e_2 \mapsto 3 \end{array} \right]$$

$$\frac{1 : \text{INT} \qquad 2 \text{ plus } 3 : \text{INT}}{1 \text{ plus } 2 \text{ plus } 3 : \text{INT}} \quad (t\text{-plus})$$

# Recursive Typing Rules

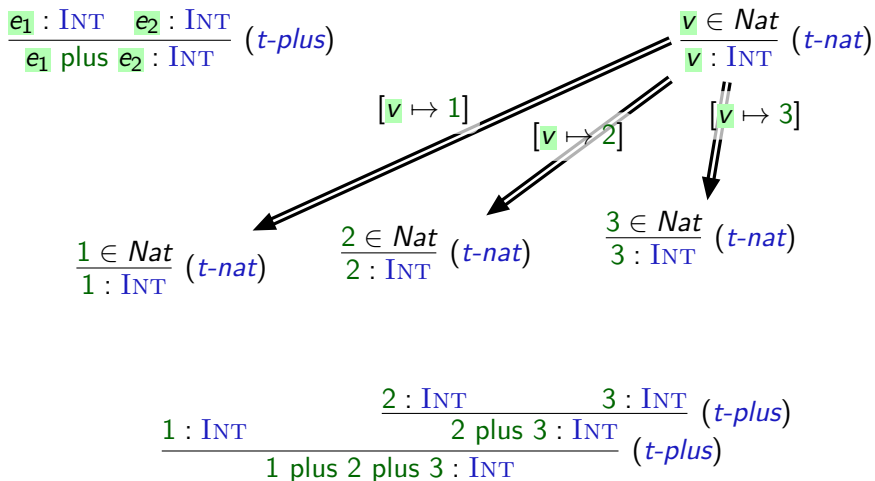
$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$



$$\frac{1 : \text{INT} \quad \frac{2 : \text{INT} \quad 3 : \text{INT}}{2 \text{ plus } 3 : \text{INT}} \quad (t\text{-plus})}{1 \text{ plus } 2 \text{ plus } 3 : \text{INT}} \quad (t\text{-plus})$$

# Recursive Typing Rules



# Recursive Typing Rules

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{\frac{1 \in \text{Nat}}{1 : \text{INT}} \quad (t\text{-nat}) \quad \frac{\frac{2 \in \text{Nat}}{2 : \text{INT}} \quad (t\text{-nat}) \quad \frac{3 \in \text{Nat}}{3 : \text{INT}} \quad (t\text{-nat})}{2 \text{ plus } 3 : \text{INT}} \quad (t\text{-plus})}{1 \text{ plus } 2 \text{ plus } 3 : \text{INT}} \quad (t\text{-plus})$$

# Types for Expressions

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true})$$

$$\frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \geq e_2 : \text{BOOL}} \quad (t\text{-ge})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \text{INT} \quad e_3 : \text{INT}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{INT}} \quad (t\text{-if-nat})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \text{BOOL} \quad e_3 : \text{BOOL}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{BOOL}} \quad (t\text{-if-bool})$$

**(*t-if*)** rule summarises (*t-if-nat*) and (*t-if-bool*) via metavariable  $\tau$

# Checking Types

► With  $e : \tau$ , we can have:

1 Exactly one  $\tau$  fits (we've computed a type):

2 plus 3 : INT

2 No  $\tau$  fits (type error):

true plus 0 *has type error*

3 Multiple  $\tau$  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:

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

# Summary

- ▶ *Type systems* relate expressions to types:

$$(\cdot) \subseteq \text{Expr} \times \mathbb{T}_{iga}$$

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

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

- ▶ No type matches  $\Rightarrow$  type error

# Question

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

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

## 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

$Expr ::= \langle Val \rangle$   
|  $id$  **new!**  
|  $let\ id = \langle Expr \rangle\ in\ \langle Expr \rangle$  **new!**  
|  $\langle Expr \rangle\ plus\ \langle Expr \rangle$   
|  $\langle Expr \rangle\ >= \langle Expr \rangle$   
|  $if\ \langle Expr \rangle\ then\ \langle Expr \rangle\ else\ \langle Expr \rangle$

$Val ::= nat$   
|  $true$  |  $false$

$nat ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \dots$

$id ::= \underline{x} \mid \underline{y} \mid \underline{z} \mid \dots$

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

# Typing Variables

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true})$$

$$\frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 >= e_2 : \text{BOOL}} \quad (t\text{-ge})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

- ▶ Same types as before:  $\mathbb{T}_{\text{inga}} = \{\text{BOOL}, \text{INT}\}$
- ▶ Need new typing rules for **let** and variables:

$$\frac{}{x : \tau} \quad (t\text{-var})$$

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

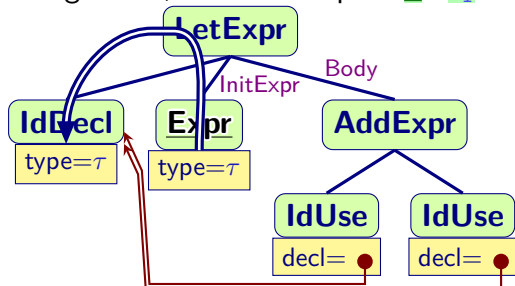
How do we connect  $\tau_1$  and  $\tau$  and  $\tau_2$  ?

# Connecting Variables and Types

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

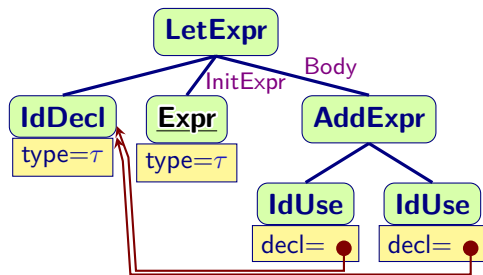
Example:  $\text{let } \underline{x} = 1 \text{ in } \underline{x} + \underline{x}$

- 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* ( $\Rightarrow$  “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

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

- ▶ Assume: name analysis does not depend on type analysis  
     $\implies$  Can identify each name by its declaration (`decl()`)
  - ▶ Formalising:
    - ▶ Write  $\Delta(\underline{x}) = \tau$  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  $\tau$
- $\implies$  *Collection Attribute*, cf. prep video for next lecture

# Typing INGA

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true}) \qquad \frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false}) \qquad \frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus}) \qquad \frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 >= e_2 : \text{BOOL}} \quad (t\text{-ge})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

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

# Example

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true})$$

$$\frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 >= e_2 : \text{BOOL}} \quad (t\text{-ge})$$

$$\frac{\Delta(\underline{x}) = \tau}{\underline{x} : \tau} \quad (t\text{-var})$$

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

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

$$\Delta(\underline{x}) = \text{INT}$$

$$\frac{\frac{1 \in \text{Nat}}{1 : \text{INT}} \quad (t\text{-nat}) \quad \Delta(\underline{x}) = \text{INT} \quad \frac{\frac{\Delta(\underline{x}) = \text{INT}}{\underline{x} : \text{INT}} \quad (t\text{-var}) \quad \frac{\Delta(\underline{x}) = \text{INT}}{\underline{x} : \text{INT}} \quad (t\text{-var})}{\underline{x} \text{ plus } \underline{x} : \text{INT}} \quad (t\text{-plus})}{\text{let } \underline{x} = 1 \text{ in } \underline{x} \text{ plus } \underline{x} : \text{INT}} \quad (t\text{-let})$$

# Summary

- ▶ Type analysis of realistic programs requires name analysis
- ▶ Distinguish if name analysis  $\leftrightarrow$  type analysis dependency and/or recursive definitions allowed:
  - 1 name  $\leftrightarrow$  type analysis, forbid recursive definitions:
    - $\implies$  Straightforward (see earlier slides)
  - 2 name  $\leftrightarrow$  type analysis, allow recursive definitions:
    - $\implies$  Requires *collecting constraints*, using  $\Delta$
  - 3 name  $\leftrightarrow$  type analysis:
    - $\implies$  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  $\Delta(x) = \tau$ : collect arbitrarily complex constraints
  - ▶ Instead of  $\Delta$ , use *environments*  $\Gamma$ 
    - ▶ Original (and “standard”) formalisation (Hindley/Damas/Milner)
    - ▶ More complex: must recursively “thread”  $\Gamma$  through typing rules
    - ▶ solves name analysis  $\leftrightarrow$  type analysis dependency
    - ▶ supports some more advanced type analyses

# Adding Lists: The Language LINGA

```
Expr ::= <Val>
      | id
      | let id = <Expr> in <Expr>
      | nil new!
      | cons (<Expr>, <Expr>) new!
      | <Expr> plus <Expr>
      | <Expr> >= <Expr>
      | if <Expr> then <Expr> else <Expr>

Val ::= nat
     | true | false
```

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

```
cons(0, cons(1, cons(2, nil)))
```

# The Type of Lists

The language of types

$\mathbb{T}_{linga}$  has one new  
production:

$$\begin{array}{l} T_y ::= \text{INT} \\ \quad | \text{BOOL} \\ \quad | \text{LIST } [\langle T_y \rangle] \end{array}$$

## Example types:

- ▶  $\text{cons}(\text{true}, \text{nil}) : \text{LIST}[\text{BOOL}]$
- ▶  $\text{cons}(1, \text{cons}(2, \text{nil})) : \text{LIST}[\text{INT}]$
- ▶  $\text{cons}(1, \text{cons}(\text{false}, \text{nil})) : \text{type error}$
- ▶  $\text{cons}(\text{cons}(1, \text{nil}), \text{nil}) : \text{LIST}[\text{LIST}[\text{INT}]]$
- ▶  $\text{nil} : \text{LIST}[\text{INT}]$   
 $\text{LIST}[\text{BOOL}]$   
 $\text{LIST}[\text{LIST}[\text{INT}]]$   
 $\text{LIST}[\dots, \text{LIST}[\text{BOOL}], \dots]$

**First attempt at typing rules:**

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

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

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

# Principal Types

- ▶  $p : \tau$  may have infinitely many  $\tau$ , can't process all
- ▶ General approach: Find **Principal Type**
  - ⇒ One *single* type that subsumes all possible types
- ▶ Extend **type language** to express such principal types
- ▶ Various approaches:
  - ▶ *Parametric Polymorphism*, using parametric types
  - ▶ Subtype Polymorphism, using subtypes
  - ▶ Type Classes
  - ...
- ▶ Here: use **Parametric Types** with **Type Variables**:
  - ▶  $\text{LIST}[\alpha]$  summarises  $\text{LIST}[\text{INT}]$ ,  $\text{LIST}[\text{BOOL}]$ ,  $\text{LIST}[\text{LIST}[\dots]]$

# Type Variables

$$\begin{array}{l} Ty ::= \text{INT} \\ \quad | \text{BOOL} \\ \quad | \text{LIST } [\langle Ty \rangle] \\ \quad | \text{tyvar} \end{array}$$
$$\text{tyvar} ::= \alpha \mid \beta \mid \gamma \mid \dots$$

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

$$\frac{}{\text{nil} : \text{LIST}[\alpha]} \quad (t\text{-nil})$$

- ▶ 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
  - ▶ `LIST[LIST[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:**
  - ▶ `LIST[ $\alpha$ ]`
  - ▶ Use **Type Variable**  $\alpha$  to express that we know the type only *partially*
- ▶ **Principal Types:**
  - ▶  $\tau$  is *principal* for  $e$  if it *subsumes* all  $\tau'$  with  $e : \tau'$  (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
  - ▶  $\mathbb{T}_{linga}$  types

## Programs: LINGA

- ▶ “Object Language” #1
- ▶ Variables refer to input programs
- ▶ Example:  $\underline{x}$  in

let  $\underline{x} = 1$  in  $\underline{x}$

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

- ▶ “Object Language” #2
- ▶ Variables refer to unknown types
- ▶ Example:  $\alpha$  in

LIST[ $\alpha$ ]

Meta-Variables Can Reference Object-Language Variables

Meta-Variable references	Example	Meta-Variable Notation
Program	1 plus 2	$\underline{e}$
Type	LIST[BOOL]	$\underline{\tau}$
Program variable	<u>foo</u>	$\underline{x}$
Type Variable	$\alpha$	$\alpha$

# 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

<https://cs.lth.se/EDAP15>