TYPE INFERENCE

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material used in this class was taken from"The Essence of ML Type Inference", a chapter written by Pottier and Rémy, available in B. Pierce's book "Advanced Topics in Types and Programming Languages"

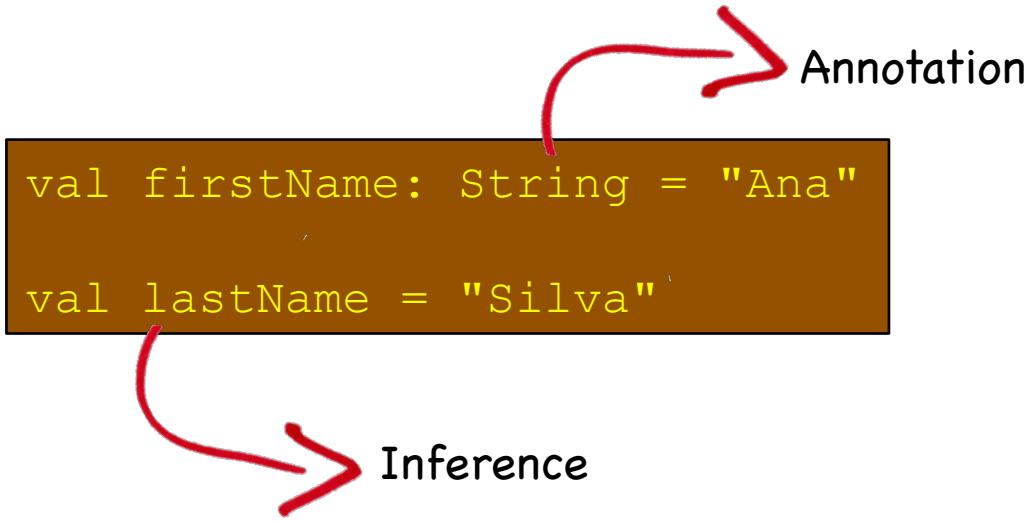
PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Type Inference

Type inference is the problem of detecting what is the type of an expression in a program.

1. Which programming languages have type inference?
2. Why is type inference useful?

Example: Kotlin



But Inference in Kotlin is still Limited

```
fun square(x: Int): Int {return x * x}

fun cube(x:Int) = square(x)*x

fun cube(x) = square(x)*x
error: a type annotation is required on a
value parameter
```

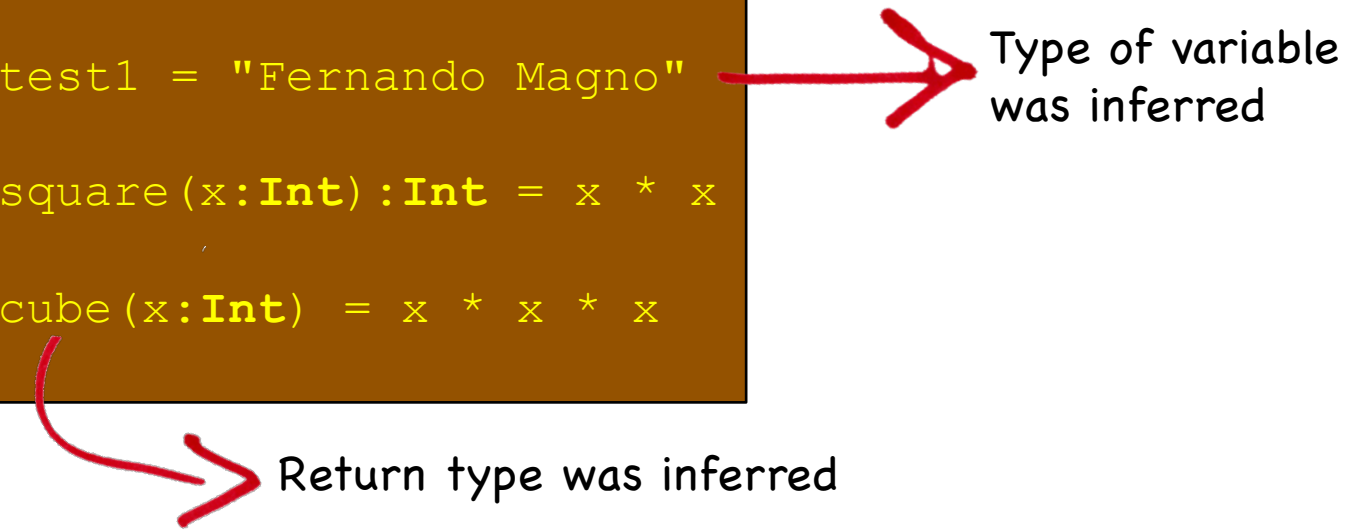
Example: Scala

```
object TestObject {

  val test1 = "Fernando Magno"

  def square(x:Int):Int = x * x

  def cube(x:Int) = x * x * x
}
```



Scala's limits are similar to Kotlin's

```
object TestObject {
  def cube(x) = x * x * x
  ':' expected, but ')' found
}
```

Why do both kotlin and scala require type annotations in function arguments?

The issue with overloading

```
object TestObject {
  def square(x:Int) = x * x
  def square(x:Double) = x * x
  def square(x:Long) = x * x
  def cube(x) = square(x) * x
}
```

1. What should be the type of 'x' in cube?
2. There are programming languages that make much more extensive use of type inference. Do you know which ones?

Type Inference in ML

```
- fun square(x) = x * x;
val square = fn : int -> int
- fun sqIf(a, b) = if a then square(b) else 0;
val sqIf = fn : bool * int -> int
- fun applyTwice(f, x) = f(f(x));
val applyTwice = fn : ('a -> 'a) * 'a -> 'a
```

1. What is the meaning of the syntax `'a`?
2. What is the type of function apply2fun?
3. How can we use apply2fun?

```
fun apply2fun(f, g, x) = f(g(x))
```

fun apply2fun(f,g,x) = f(g(x))

- Assume that `x` has a type 'c and g has a type `'t -> 'a`

What do we know about `'c` and `'x`?

fun apply2fun(f,g,x) = f(g(x))

- Assume that `x` has a type 'c and g has a type `'t -> 'a`

What do we know about `'c` and `'t`?

- The types `'c` and `'t` must be the same!
  - Thus, the type of g is `'c -> 'a`
- Assume now that `f` has a type `s' -> 'b`

What do we know about `'a` and `'s`?

fun apply2fun(f,g,x) = f(g(x))

- Assume that `x` has a type 'c and g has a type `'t -> 'a`

What do we know about `'c` and `'t`?

- The types `'c` and `'t` must be the same!
  - Thus, the type of g is `'c -> 'a`
- Assume now that `f` has a type `s' -> 'b`

What do we know about `'a` and `'s`?

- The types `'a` and `'s` must be the same!
  - Thus, the type of f is `'a -> 'b`

What's then the type of `apply2fun`?

fun apply2fun(f,g,x) = f(g(x))

```
- fun apply2fun(f, g, x) = f(g(x));
val apply2fun = fn : ('a -> 'b) * ('c -> 'a) * 'c -> 'b
```

If you had to design a type-inference algorithm, how would you do it?

Constraint-Based Analysis

- Generate constraints
- Solve the constraints
- Annotate the program

What does "correctness" mean for such an algorithm?

The Correctness Criterion

- Generate constraints
- Solve the constraints
- Annotate the program

The type-inference algorithm is correct if we can type-check the program annotated with the inferred types.

What does it mean to "type-check" a program?

Fernando Magno Quintão Pereira

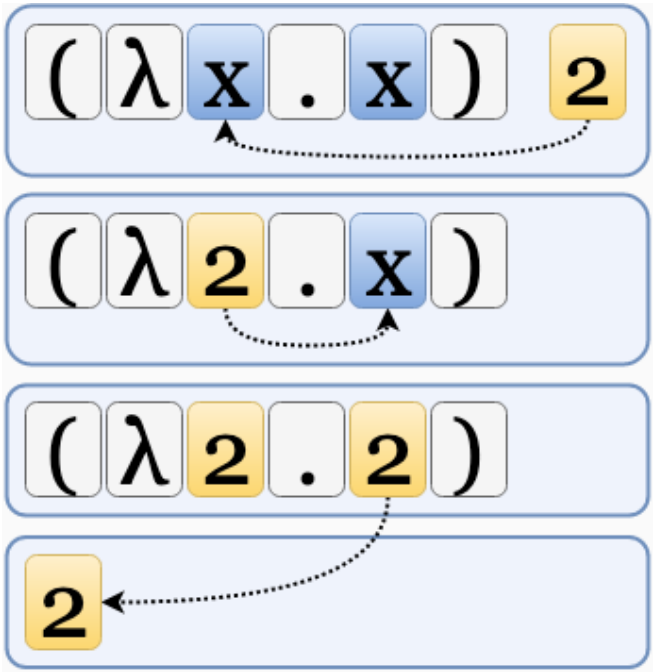
fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Simply Typed Lambda Calculus



fernando@dcc.ufmg.br

The Simply Typed Lambda Calculus

Syntax

T ::=
- bool
- int
- T → T

E ::=
- x
- λx:T.E
- E E
- *c*

What does this syntax mean?

Equivalence in ML

Syntax

T ::=	E ::=
bool	x
int	λx:T.E
T → T	E E
	c

What are the equivalences between our syntax and ML's in each expression below?

```
1;
true;
(fn x:int => x + x);
v;
(fn x:int => x * x) 4
(fn y: int->int => fn x:int => y x) (fn a:int => a * a);
```

e.g., after declaring `val v = true`

Equivalent syntax in ML

Syntax

T ::=	E ::=
bool	x
int	λx:T.E
T → T	E E
	c

Can you write typing rules for each syntactic rule of the simply typed lambda calculus?

c	`1;`
c	`true;`
λx:T.E	`(fn x:int => x + x);`
x	`v;` e.g., after declaring `val v = true`
E E	`(fn x:int => x * x) 4`
E E	`(fn y: int->int => fn x:int => y x) (fn a:int => a * a);`

Notice that the type declarations are optional in every example above

Typing Rules

Syntax

T ::=
- bool
- int
- T → T

E ::=
- x
- λx:T.E
- E E
- *c*

Typing Rules

$$\text{true: bool} \qquad \text{[T-TRUE]}$$

$$\text{false: bool} \qquad \text{[T-FALSE]}$$

$$\text{n: int} \qquad \text{[T-NAT]}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T} \qquad \text{[T-VAR]}$$

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T_1.E)\colon T_1 \to T_2} \qquad \text{[T-ABS]}$$

$$\frac{\Gamma \vdash E_1{:}T \to T_1 \qquad \Gamma \vdash E_2{:}T}{\Gamma \vdash E_1 E_2\colon T_1} \qquad \text{[T-APP]}$$

Ex.: $((\lambda y{:}T_1 \rightarrow T_2.(\lambda x{:}T_1.(y\ x)))\ (\lambda a{:}T_3.a))\ 1$

1. What's the value of the expression above?
2. Can you find its type?

Ex.: $((\lambda y{:}T_1 \rightarrow T_2.(\lambda x{:}T_1.(y\ x)))\ (\lambda a{:}T_3.a))\ 1$

1. What's the value of the expression above?
2. Can you find its type?

```
(fn y => fn x => y x) (fn a => a) 1;
val it = 1 : int
```

$((\lambda y{:}T_1 \to T_2.(\lambda x{:}T_1.(y\ x)))\ (\lambda a{:}T_3.a))\ 1$

Typing Rules

$$\text{true: bool} \qquad \text{[T-TRUE]}$$

$$\text{false: bool} \qquad \text{[T-FALSE]}$$

$$\text{n: int} \qquad \text{[T-NAT]}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T} \qquad \text{[T-VAR]}$$

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T.E){:}\ T_1 \to T_2} \qquad \text{[T-ABS]}$$

$$\frac{\Gamma \vdash E_1{:}T \to T_1 \qquad \Gamma \vdash E_2{:}T}{\Gamma \vdash E_1 E_2{:}\ T_1} \qquad \text{[T-APP]}$$

What's the first rule that applies?

$((\lambda y{:}T_1 \to T_2.(\lambda x{:}T_1.(y\ x)))\ (\lambda a{:}T_3.a))\ 1$

Typing Rules

$$\text{true: bool} \qquad \text{[T-TRUE]}$$

$$\text{false: bool} \qquad \text{[T-FALSE]}$$

$$\text{n: int} \qquad \text{[T-NAT]}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T} \qquad \text{[T-VAR]}$$

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T.E){:}\ T_1 \to T_2} \qquad \text{[T-ABS]}$$

$$\frac{\Gamma \vdash E_1{:}T \to T_1 \qquad \Gamma \vdash E_2{:}T}{\Gamma \vdash E_1 E_2{:}\ T_1} \qquad \text{[T-APP]}$$

What are E_1 and E_2 in this case?

The Derivation Tree

Each line represents the application of one of the rules (T-TRUE, T-FALSE, T-NAT, T-VAR, T-ABS, T-APP). Can you identify each rule?

$$\dfrac{\dfrac{\dfrac{\dfrac{y{:}int\to int \in \Gamma, y{:}int\to int, x{:}int}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y: int\to int} \quad \dfrac{x{:}int \in \Gamma, y{:}int\to int, x{:}int}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x: int}}{\Gamma, y{:}int\to int, x{:}int \vdash y\ x: int\to int}}{\Gamma, y{:}int\to int \vdash (\lambda x{:}int.(y\ x)): int\to int} \quad \dfrac{\dfrac{a{:}int \in \Gamma, a{:}int}{\Gamma, a{:}int \vdash a: int}}{\Gamma \vdash \lambda a{:}int.a: int\to int}}{\Gamma \vdash ((\lambda y{:}int\to int.(\lambda x{:}int.(y\ x))): (int\to int)\to int} \quad \dfrac{}{\Gamma \vdash ((\lambda y{:}int\to int.(\lambda x{:}int.(y\ x)))\ (\lambda a{:}int.a)): int\to int} \quad \dfrac{}{1 : int}}{\Gamma \vdash ((\lambda y{:}int\to int.(\lambda x{:}int.(y\ x)))\ (\lambda a{:}int.a))\ 1 : int}$$

The Derivation Tree

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T}\ [\text{T-VAR}]$$

$$\frac{y{:}\text{int}\to\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y{:}\ \text{int}\to\text{int}}$$

The Derivation Tree

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T}\ [\text{T-VAR}]$$

$$\frac{y{:}int\to int \in \Gamma, y{:}int\to int, x{:}int}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y{:}\ int\to int} \qquad \frac{x{:}int \in \Gamma, y{:}int\to int, x{:}int}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x{:}\ int}$$

The Derivation Tree

$$\frac{\Gamma \vdash E_1 : T \to T_1 \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 E_2 : T_1}\ [\text{T-App}]$$

$$\frac{\dfrac{y{:}\text{int}\to\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y : \text{int}\to\text{int}} \qquad \dfrac{x{:}\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x : \text{int}}}{\Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int} \vdash y\ x : \text{int}}$$

The Derivation Tree

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T_1.E)\colon T_1 \to T_2}\ [\text{T-ABS}]$$

$$\frac{\dfrac{y{:}\text{int}\to\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y\colon \text{int}\to\text{int}} \qquad \dfrac{x{:}\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x\colon \text{int}}}{\Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int} \vdash y\ x\colon \text{int}}$$

$$\Gamma, y{:}\text{int}\to\text{int} \vdash (\lambda x{:}\text{int}.(y\ x))\colon \text{int}\to\text{int}$$

The Derivation Tree

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T}\ [\text{T-VAR}]$$

$$\frac{\dfrac{y{:}\text{int}\to\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y{:}\ \text{int}\to\text{int}} \qquad \dfrac{x{:}\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x{:}\ \text{int}}}{\dfrac{\Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int} \vdash y\ x{:}\ \text{int}}{\Gamma, y{:}\text{int}\to\text{int} \vdash (\lambda x{:}\text{int}.(y\ x)){:}\ \text{int}\to\text{int}}} \qquad \frac{a{:}\text{int} \in \Gamma, a{:}\text{int}}{\Gamma,\ a{:}\text{int} \vdash a{:}\ \text{int}}$$

The Derivation Tree

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T_1.E)\colon T_1 \to T_2}\ [\text{T-ABS}]$$

$$\dfrac{\dfrac{y{:}\text{int}\to\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y\colon \text{int}\to\text{int}} \quad \dfrac{x{:}\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x\colon \text{int}}}{\dfrac{\Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int} \vdash y\ x\colon \text{int}}{\dfrac{\Gamma, y{:}\text{int}\to\text{int} \vdash (\lambda x{:}\text{int}.(y\ x))\colon \text{int}\to\text{int}}{\Gamma \vdash ((\lambda y{:}\text{int}\to\text{int}.(\lambda x{:}\text{int}.(y\ x)))\colon (\text{int}\to\text{int})\to\text{int}\to\text{int}}}} \qquad \dfrac{a{:}\text{int} \in \Gamma, a{:}\text{int}}{\Gamma, a{:}\text{int} \vdash a\colon \text{int}}$$

The Derivation Tree

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T_1.E)\colon T_1 \to T_2}\ [\text{T-ABS}]$$

$$\frac{\dfrac{\dfrac{y{:}\text{int}\to\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y\colon \text{int}\to\text{int}} \quad \dfrac{x{:}\text{int} \in \Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int}}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x\colon \text{int}}}{\Gamma, y{:}\text{int}\to\text{int}, x{:}\text{int} \vdash y\ x\colon \text{int}}}{\Gamma, y{:}\text{int}\to\text{int} \vdash (\lambda x{:}\text{int}.(y\ x))\colon \text{int}\to\text{int}} \quad \dfrac{a{:}\text{int} \in \Gamma, a{:}\text{int}}{\Gamma, a{:}\text{int} \vdash a\colon \text{int}}}{\Gamma \vdash ((\lambda y{:}\text{int}\to\text{int}.(\lambda x{:}\text{int}.(y\ x)))\colon (\text{int}\to\text{int})\to\text{int}\to\text{int}} \quad \frac{}{\Gamma \vdash \lambda a{:}\text{int}.a\colon \text{int}\to\text{int}}$$

The Derivation Tree

$$\frac{\Gamma \vdash E_1 : T \to T_1 \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 E_2 : T_1}\ [\text{T-App}]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{y:int\to int \in \Gamma, y:int\to int, x:int}{\Gamma, y:T_1\to T_2, x:T_1 \vdash y: int\to int} \qquad \dfrac{x:int \in \Gamma, y:int\to int, x:int}{\Gamma, y:T_1\to T_2, x:T_1 \vdash x: int}}{\Gamma, y:int\to int, x:int \vdash y\ x: int}}{\Gamma, y:int\to int \vdash (\lambda x:int.(y\ x)): int\to int} \qquad \dfrac{\dfrac{a:int \in \Gamma, a:int}{\Gamma, a:int \vdash a: int}}{\Gamma \vdash \lambda a:int.a: int\to int}}{\Gamma \vdash ((\lambda y:int\to int.(\lambda x:int.(y\ x))): (int\to int)\to int\to int}$$

$$\Gamma \vdash ((\lambda y:int\to int.(\lambda x:int.(y\ x)))\ (\lambda a:int.a)): int\to int$$

The Derivation Tree

$$\frac{\Gamma \vdash E_1 : T \to T_1 \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 E_2 : T_1}\ [\text{T-App}]$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{y{:}int\to int \in \Gamma, y{:}int\to int, x{:}int}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash y : int\to int} \qquad \dfrac{x{:}int \in \Gamma, y{:}int\to int, x{:}int}{\Gamma, y{:}T_1\to T_2, x{:}T_1 \vdash x : int}}{\Gamma, y{:}int\to int, x{:}int \vdash y\ x : int}}{\Gamma, y{:}int\to int \vdash (\lambda x{:}int.(y\ x)) : int\to int} \qquad \dfrac{\dfrac{a{:}int \in \Gamma, a{:}int}{\Gamma, a{:}int \vdash a : int}}{\Gamma \vdash \lambda a{:}int.a : int\to int}}{\Gamma \vdash ((\lambda y{:}int\to int.(\lambda x{:}int.(y\ x))) (\lambda a{:}int.a)) : int\to int} \qquad 1 : int}{\Gamma \vdash ((\lambda y{:}int\to int.(\lambda x{:}int.(y\ x))) (\lambda a{:}int.a))\ 1 : int}$$

The Augmented Language

Syntax

T ::=
- bool
- int
- T → T

E ::=
- x
- λx:T.E
- E E
- *c*
- E + E
- E < E
- E and E
- if E then E else E

Can you think about type-checking rules for these new expressions?

We shall add four new expressions, just to make type-checking (and inference) more interesting

The New Type-Checking Rules

true: bool [T-TRUE]

false: bool [T-FALSE]

n: int [T-NAT]

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T}$$ [T-VAR]

$$\frac{\Gamma, x{:}T_1 \vdash E{:}T_2}{\Gamma \vdash (\lambda x{:}T_1.E)\colon T_1 \to T_2}$$ [T-ABS]

$$\frac{\Gamma \vdash E_1{:}T \to T_1 \qquad \Gamma \vdash E_2{:}T}{\Gamma \vdash E_1 E_2\colon T_1}$$ [T-APP]

$$\frac{\Gamma \vdash E_1\colon \text{int} \qquad \Gamma \vdash E_2\colon \text{int}}{\Gamma \vdash E_1 + E_2\colon \text{int}}$$ [T-ADD]

$$\frac{\Gamma \vdash E_1\colon \text{int} \qquad \Gamma \vdash E_2\colon \text{int}}{\Gamma \vdash E_1 < E_2\colon \text{bool}}$$ [T-LTH]

$$\frac{\Gamma \vdash E_1\colon \text{bool} \qquad \Gamma \vdash E_2\colon \text{bool}}{\Gamma \vdash E_1 \text{ and } E_2\colon \text{bool}}$$ [T-AND]

$$\frac{\Gamma \vdash E_1\colon \text{bool} \qquad \Gamma \vdash E_2\colon T \qquad \Gamma \vdash E_3\colon T}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : T}$$ [T-IF]

The Type Inference Problem

Given a program with open type variables instead of concrete types, e.g.:

$((\lambda y{:}T_1.(\lambda x{:}T_2.(y\ x)))\ (\lambda a{:}T_3.a))\ 1$

Can you rewrite it, replacing type variables with concrete types, e.g.:

$((\lambda y{:}\text{int}\rightarrow\text{int}.(\lambda x{:}\text{int}.(y\ x)))\ (\lambda a{:}\text{int}.a))\ 1$

so that the resulting program type checks?

Solution:

$T_1 = \text{int}\rightarrow\text{int}$

$T_2 = \text{int}$

$T_3 = \text{int}$

Again, what does it mean for a program to "type check"?

The Type Checking Problem

Given a program P, e.g.:

P = ((λy:int→int.(λx:int.(y x))) (λa:int.a)) 1

of type T, and an environment Γ that assigns types to the *free variables* in P, is it the case that $\Gamma \vdash P : T$?

What are "free variables"?

Free Variables

Free variables are the variables used, but not declared in a program.

$fv(true) = \emptyset$

$fv(false) = \emptyset$

$fv(n) = \emptyset$

$fv(x) = \{x\}$

$fv(\lambda x{:}T.E) = fv(E) - \{x\}$

$fv(E_1 E_2) = fv(E_1) \cup fv(E_2)$

$fv(E_1 + E_2) = fv(E_1) \cup fv(E_2)$

$fv(E_1 < E_2) = fv(E_1) \cup fv(E_2)$

$fv(E_1 \text{ and } E_2) = fv(E_1) \cup fv(E_2)$

$fv(\text{if } E_1 \text{ then } E_2 \text{ else } E_3) = fv(E_1) \cup fv(E_2) \cup fv(E_3)$

Why is it necessary to account for the free variables in the statement of the type checking problem?

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Constraint-Based Type Inference

fernando@dcc.ufmg.br

Back into Constraint-Based Analysis

- Generate constraints
- Solve the constraints
- Annotate the program

1. What are constraints?
2. How do we generate them?

Constraint Generation

- We shall define a function to generate constraints
- K = gen(Γ, E, τ)
  - Γ is an environment that associates variables with *type variables*
  - E is the expression that we shall parse
  - τ is a type variable that will hold the type of expression E
  - K are the constraints that we shall generate

1. We say that constraint generation is syntax directed. What does that mean?
2. What would be a type variable?

Type Variables

- A type variable is a placeholder for a concrete type. We shall denote them by α (alpha) and τ (tau).

$\alpha_1 \rightarrow \text{int} \rightarrow \text{int}$

$\alpha_1 \rightarrow \alpha_2 \rightarrow \text{bool}$

$\text{bool} \rightarrow \tau_1 \rightarrow \alpha_2$

$\tau_1 \rightarrow \tau_2$

- We shall reserve τ for the type of program variables, or the type of the entire program:

$\lambda x{:}\tau_1.(\lambda y{:}\tau_2.\ x + y) : \tau_3$

$\lambda x{:}\tau_1.(\lambda y{:}\tau_2.(\lambda z{:}\tau_3.\ \text{if } x(y) \text{ then } z \text{ else } z + 1)) : \tau_4$

What would be the type of τ_1, τ_2, τ_3 (and τ_4) so that these programs type-check?

The Language of Constraints

- Constraints are conjunctions of equivalences:

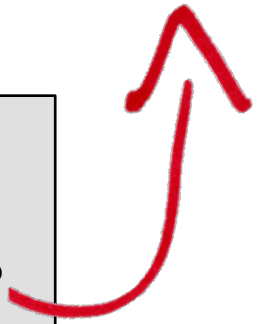
K ::=
- K ∧ K
- T ≡ T

T ::=
- bool
- int
- T → T
- α
- τ

Example:

$K = \tau_0 \equiv \tau_2 \to \alpha_1 \wedge \alpha_2 \to \alpha_1 \equiv \tau_1 \to \alpha_3 \wedge \alpha_3 \equiv \tau_1 \wedge \alpha_2 \equiv \tau_2$

How many constraints do we have here?



Ground Types

- A ground type is a type without unquantified type variables:

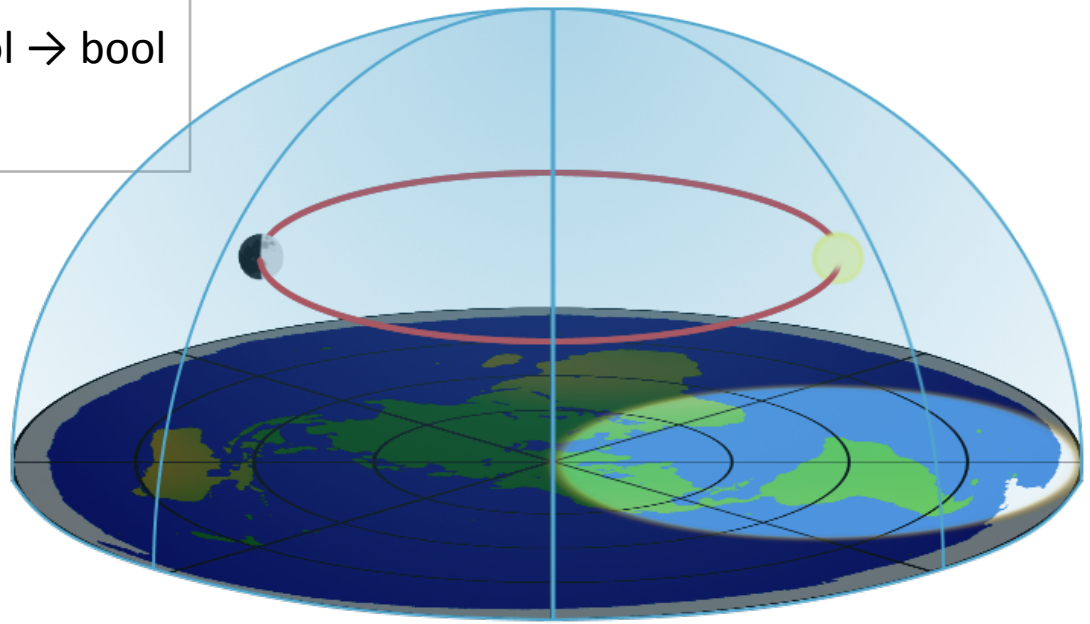
What's a (un)quantified typed variable?

bool → int → int

int → int → bool

bool → bool → bool

int → int



Ground truth

Parametric Polymorphism

- We can leave a few type variables open, and the program still type-checks:

$\forall \tau.\lambda x{:}\tau \rightarrow bool.(\lambda y{:}\tau.(\lambda z{:}int.\ \text{if } x(y) \text{ then } z \text{ else } z + 1)) : int$

- $\forall \tau.\tau \rightarrow bool$ means that this type involves any function that returns bool, e.g.:

```
- val f = fn x => fn y => fn z => if (x(y)) then z else z + 1;
val f = fn : ('a -> bool) -> 'a -> int -> int
- val g1 = fn x => x > 0;
val g1 = fn : int -> bool
- f g1 3 5;
val it = 5 : int
- val g2 = fn x => x andalso true;
val g2 = fn : bool -> bool
- f g1 4 5;
val it = 5 : int
```

Type Inference and Ground Types

Another way to see the type-inference problem: replace type variables with ground types, so that the resulting program type-checks.

$\lambda x{:}\tau_1.(\lambda y{:}\tau_2.(\lambda z{:}\tau_3.\ \text{if } x(y) \text{ then } z \text{ else } z + 1)) : \tau_4$

Solution

$\tau_1 \mapsto \forall\tau.\tau\to\text{bool}$

$\tau_2 \mapsto \forall\tau.\tau$

$\tau_3 \mapsto \text{int}$

$\tau_4 \mapsto \forall\tau.(\tau\to\text{bool})\to\tau\to\text{int}\to\text{int}$

$\forall\tau.\lambda x{:}\tau\to\text{bool}.(\lambda y{:}\tau.(\lambda z{:}\text{int}.\ \text{if } x(y) \text{ then } z \text{ else } z + 1)) : \text{int}$

Implementing the gen function (constants)

E ::=
- x
- λx:T.E
- E E
- *c*
- E + E
- E < E
- E and E
- if E then E else E

$$\frac{c \text{ is integer literal}}{\text{gen}(\Gamma, c, \tau) = \tau \equiv \text{int}}$$

1. What is the meaning of the syntax above?
2. How would be the constraints for the other constants, e.g., true and false?

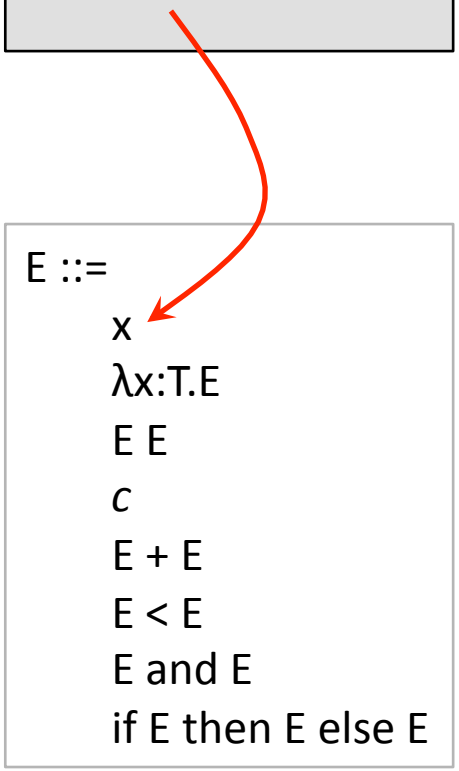
Implementing the gen function (constants)

$$\frac{c \text{ is integer literal}}{gen(\Gamma, c, \tau) = \tau \equiv int}$$

$gen(\Gamma, true, \tau) = \tau \equiv bool$

$gen(\Gamma, false, \tau) = \tau \equiv bool$

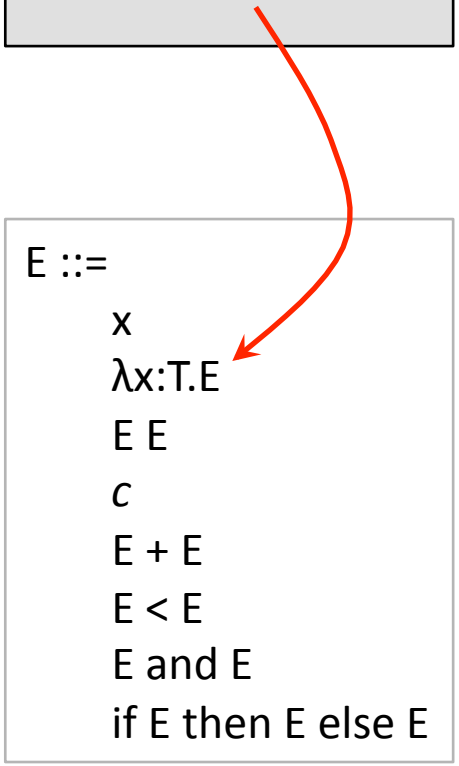
How would you generate constraints for variables?



Implementing the gen function (variables)

$$\frac{\Gamma[x] = \tau_x}{gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

How would you generate constraints for abstractions?



Implementing the gen function (abstractions)

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{\text{gen}(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \rightarrow \alpha \wedge K}$$

Lots of things happening here... can you explain them?

Implementing the gen function (abstractions)

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{\text{gen}(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K}$$

$$\frac{\Gamma[x] = \tau_x}{\text{gen}(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

It helps to check the case of variables. Got it?

Implementing the gen function (abstractions)

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{\text{gen}(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K}$$

$$\frac{\Gamma[x] = \tau_x}{\text{gen}(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

It helps to check also the case of variables. Got it?

Implementing the gen function (addition)

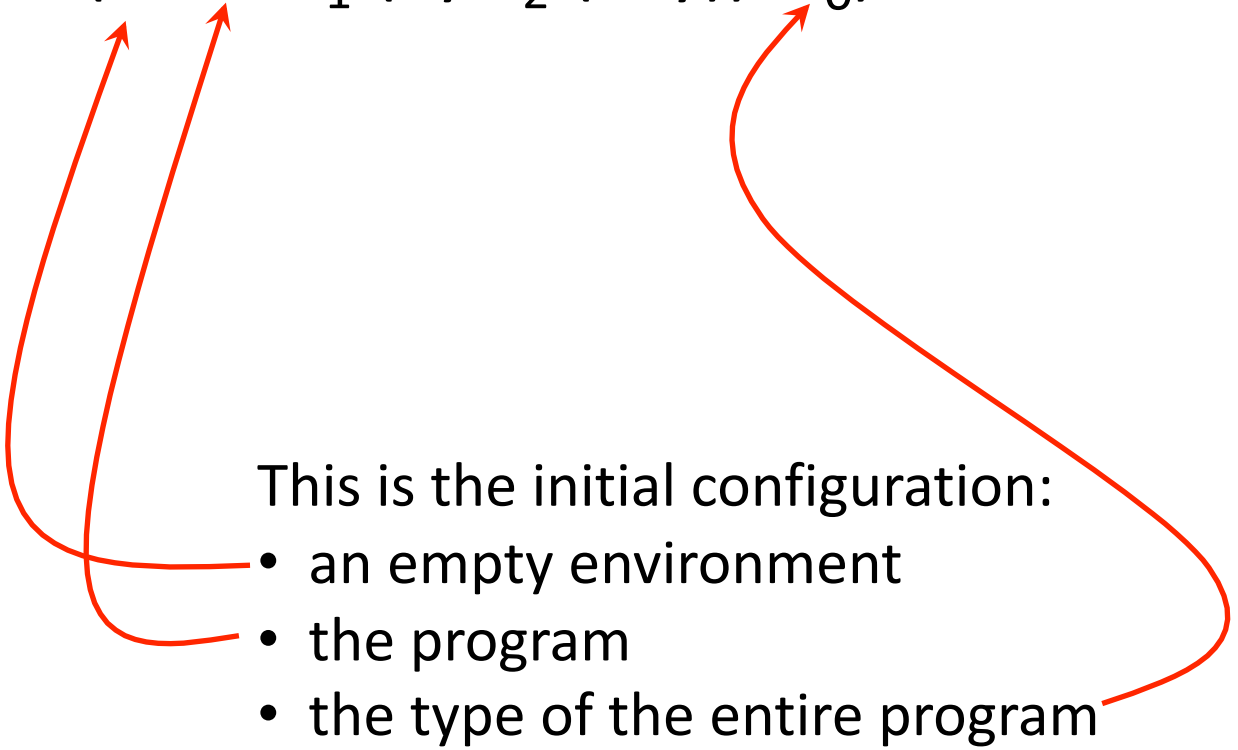
$$\frac{gen(\Gamma, E_1, int) = K_1 \qquad gen(\Gamma, E_2, int) = K_2}{gen(\Gamma, E_1+E_2, \tau) = \tau \equiv int \wedge K_1 \wedge K_2}$$

$$\frac{fresh(\alpha) \qquad gen(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{gen(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \rightarrow \alpha \wedge K}$$

$$\frac{\Gamma[x] = \tau_x}{gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

Let's try to generate all the constraints for this program: $\lambda x{:}\tau_2.(\lambda y{:}\tau_1.(x+y))$

Ex.: gen(Ø, $\lambda x{:}\tau_1.(\lambda y{:}\tau_2.(x+y))$, τ_0)



This is the initial configuration:

- an empty environment
- the program
- the type of the entire program

What's then the result of calling gen on:

$\lambda x{:}\tau_2.(\lambda y{:}\tau_1.(x+y)) : \tau_0$?

Example: gen(Ø, $\lambda x{:}\tau_1.(\lambda y{:}\tau_2.(x+y))$, τ_0)

$$\frac{\text{gen}(\Gamma, E_1, \text{int}) = K_1 \qquad \text{gen}(\Gamma, E_2, \text{int}) = K_2}{\text{gen}(\Gamma, E_1+E_2, \tau) = \tau \equiv \text{int} \wedge K_1 \wedge K_2}$$

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{\text{gen}(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \rightarrow \alpha \wedge K}$$

$$\frac{\Gamma[x] = \tau_x}{\text{gen}(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

Example: gen(Ø, $\lambda x{:}\tau_1.(\lambda y{:}\tau_2.(x+y))$, τ_0)

$$\frac{\text{gen}(\Gamma, E_1, \text{int}) = K_1 \qquad \text{gen}(\Gamma, E_2, \text{int}) = K_2}{\text{gen}(\Gamma, E_1+E_2, \tau) = \tau \equiv \text{int} \wedge K_1 \wedge K_2}$$

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{\text{gen}(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \rightarrow \alpha \wedge K} \qquad \frac{\Gamma[x] = \tau_x}{\text{gen}(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

Solution:

$K = \tau_0 \equiv \tau_2 \rightarrow \alpha_1 \wedge \alpha_1 \equiv \tau_1 \rightarrow \alpha_2 \wedge \alpha_2 \equiv \text{int} \wedge \text{int} \equiv \tau_2 \wedge \text{int} \equiv \tau_1$

By the way...

Could you solve those constraints below, to find types for the program, e.g.:
$\lambda x{:}\tau_2.(\lambda y{:}\tau_1.(x+y))$
The type of this expression is τ_0

Solution:

$K = \tau_0 \equiv \tau_2 \rightarrow \alpha_1 \wedge \alpha_1 \equiv \tau_1 \rightarrow \alpha_2 \wedge \alpha_2 \equiv \text{int} \wedge \text{int} \equiv \tau_2 \wedge \text{int} \equiv \tau_1$

Implementing the gen function (application)

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma, E_1, \alpha \to \tau) = K_1 \qquad \text{gen}(\Gamma, E_2, \alpha) = K_2}{\text{gen}(\Gamma, E_1 E_2, \tau) = K_1 \wedge K_2}$$

E ::=
- x
- λx:T.E
- E E
- *c*
- E + E
- E < E
- E and E
- if E then E else E

Can you show constraint generation for the syntactic constructs still left?

Constraints for the other expressions

$$\frac{gen(\Gamma, E_1, int) = K_1 \qquad gen(\Gamma, E_2, int) = K_2}{gen(\Gamma, E_1 < E_2, \tau) = \tau \equiv bool \wedge K_1 \wedge K_2}$$

$$\frac{gen(\Gamma, E_1, bool) = K_1 \qquad gen(\Gamma, E_2, bool) = K_2}{gen(\Gamma, E_1 \text{ and } E_2, \tau) = \tau \equiv bool \wedge K_1 \wedge K_2}$$

$$\frac{gen(\Gamma, E_1, bool) = K_1 \qquad gen(\Gamma, E_2, \tau) = K_2 \qquad gen(\Gamma, E_3, \tau) = K_3}{gen(\Gamma, \text{if } E_1 \text{ then } E_2 \text{ else } E_3, \tau) = K_1 \wedge K_2 \wedge K_3}$$

The Complete Implementation of gen

$$\frac{c \text{ is integer literal}}{gen(\Gamma, c, \tau) = \tau \equiv int}$$

$$gen(\Gamma, true, \tau) = \tau \equiv bool$$

$$gen(\Gamma, false, \tau) = \tau \equiv bool$$

$$\frac{\Gamma[x] = \tau_x}{gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

$$\frac{fresh(\alpha) \qquad gen(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{gen(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K}$$

$$\frac{gen(\Gamma, E_1, bool) = K_1 \qquad gen(\Gamma, E_2, \tau) = K_2 \qquad gen(\Gamma, E_3, \tau) = K_3}{gen(\Gamma, \text{if } E_1 \text{ then } E_2 \text{ else } E_3, \tau) = K_1 \wedge K_2 \wedge K_3}$$

$$\frac{gen(\Gamma, E_1, int) = K_1 \qquad gen(\Gamma, E_2, int) = K_2}{gen(\Gamma, E_1 + E_2, \tau) = \tau \equiv int \wedge K_1 \wedge K_2}$$

$$\frac{gen(\Gamma, E_1, int) = K_1 \qquad gen(\Gamma, E_2, int) = K_2}{gen(\Gamma, E_1 < E_2, \tau) = \tau \equiv bool \wedge K_1 \wedge K_2}$$

$$\frac{gen(\Gamma, E_1, bool) = K_1 \qquad gen(\Gamma, E_2, bool) = K_2}{gen(\Gamma, E_1 \text{ and } E_2, \tau) = \tau \equiv bool \wedge K_1 \wedge K_2}$$

$$\frac{fresh(\alpha) \qquad gen(\Gamma, E_1, \alpha \to \tau) = K_1 \qquad gen(\Gamma, E_2, \alpha) = K_2}{gen(\Gamma, E_1 E_2, \tau) = K_1 \wedge K_2}$$

gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)

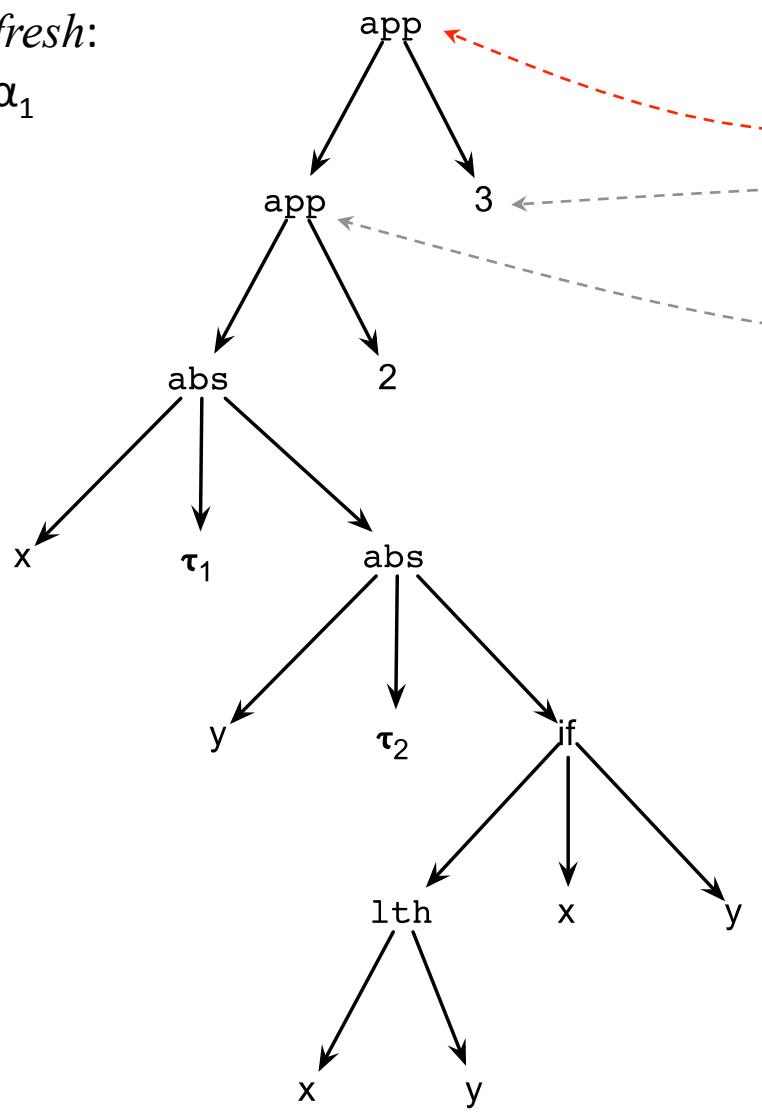
E ::=
- x
- λx:T.E
- E E
- *c*
- E + E
- E < E
- E and E
- if E then E else E

1. What is the semantics of the function above?
2. Can you try to draw its abstract syntax tree?
3. Can you run gen over this tree, to see which constraints you get?

gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)

fresh:

α_1

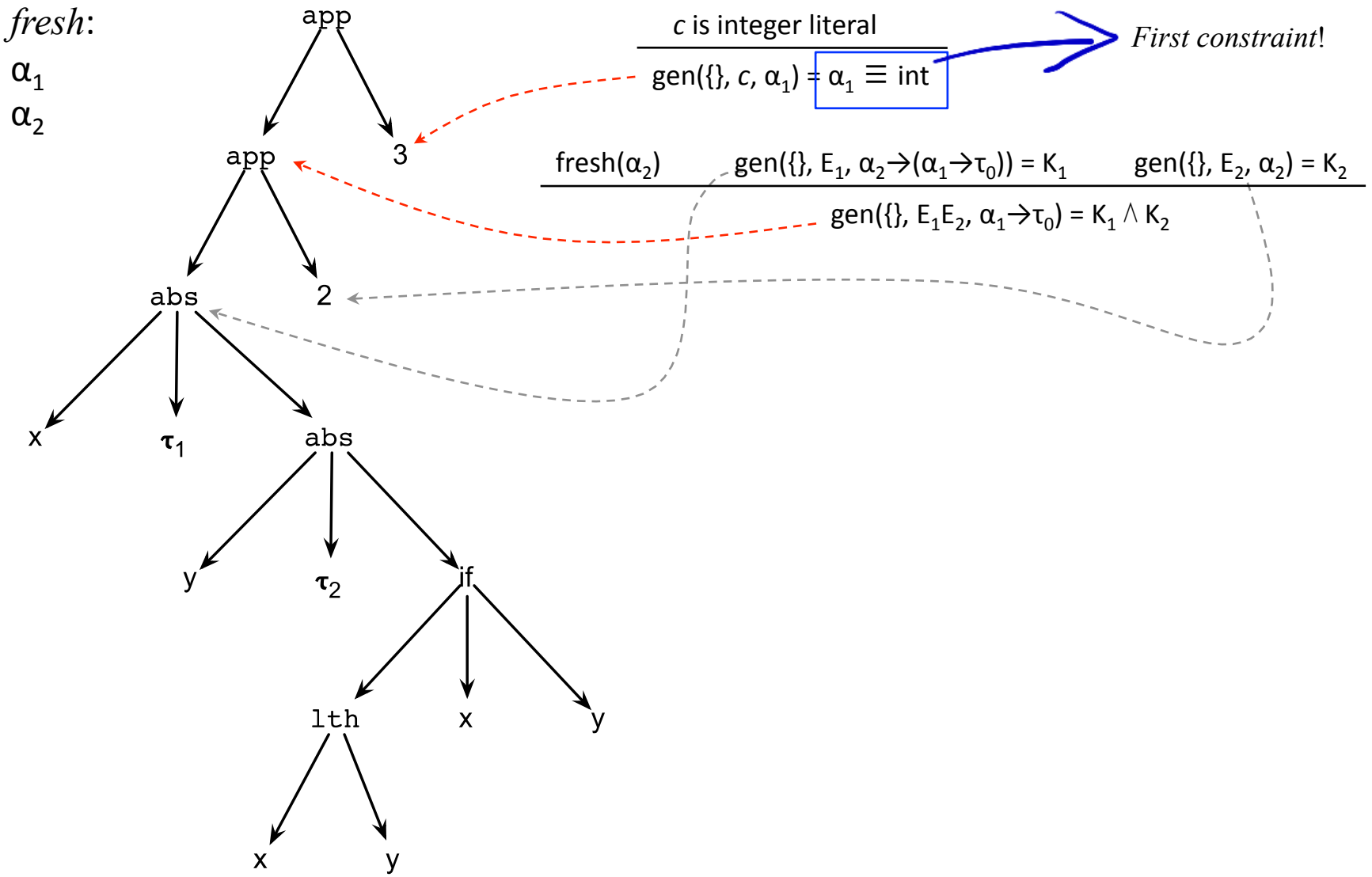


$$\frac{\text{fresh}(\alpha_1) \qquad \text{gen}(\{\}, E_1, \alpha_1 \rightarrow \tau_0) = K_1 \qquad \text{gen}(\{\}, E_2, \alpha_1) = K_2}{\text{gen}(\{\}, E_1E_2, \tau_0) = K_1 \wedge K_2}$$

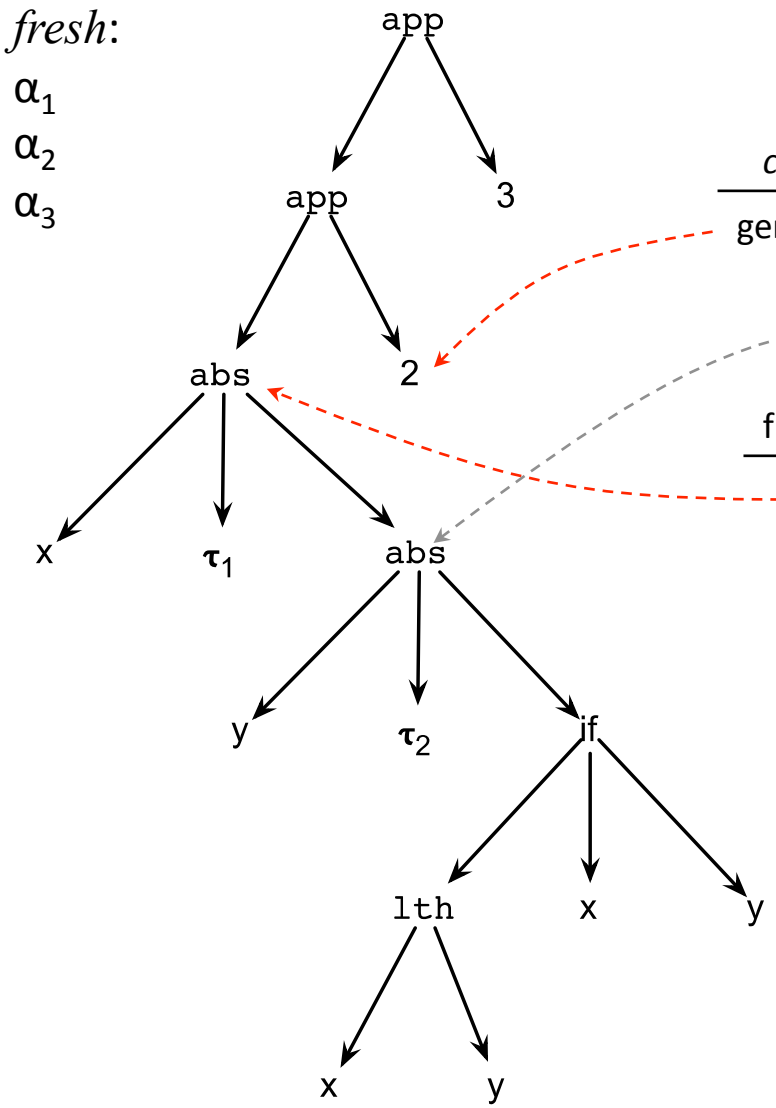
Constraint generation for applications:

$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\boldsymbol{\Gamma}, E_1, \alpha \rightarrow \tau) = K_1 \qquad \text{gen}(\boldsymbol{\Gamma}, E_2, \alpha) = K_2}{\text{gen}(\boldsymbol{\Gamma}, E_1E_2, \tau) = K_1 \wedge K_2}$$

gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)



gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)



$$\frac{c \text{ is integer literal}}{\text{gen}(\Gamma, c, \alpha_2) = \alpha_2 \equiv \text{int}}$$

Second constraint!

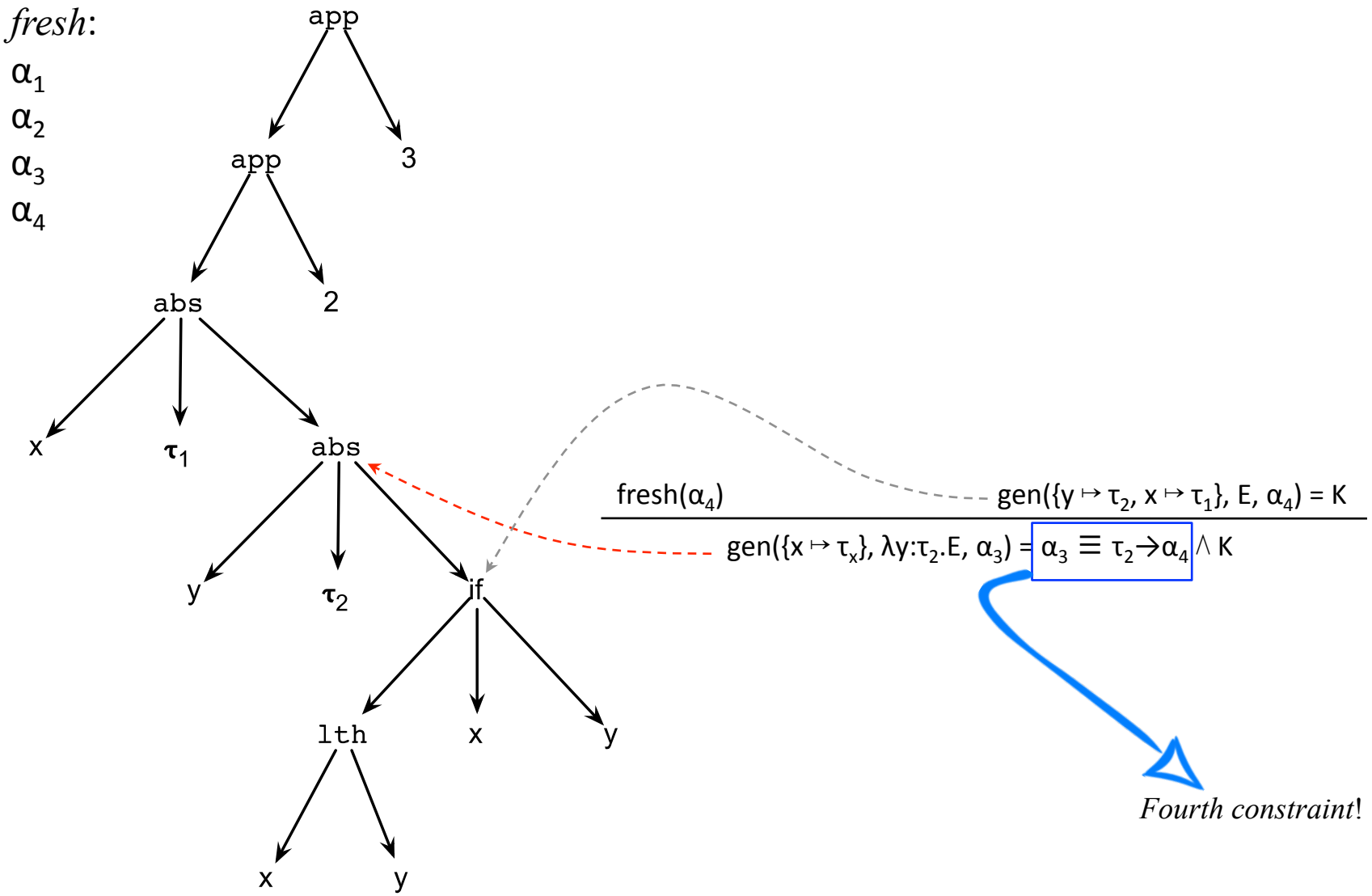
$$\frac{\text{fresh}(\alpha_3) \qquad \text{gen}(\{x \mapsto \tau_1\}, E, \alpha_3) = K}{\text{gen}(\{\}, \lambda x{:}\tau_1.E, \alpha_2 \to (\alpha_1 \to \tau_0)) = \alpha_2 \to (\alpha_1 \to \tau_0) \equiv \tau_1 \to \alpha_3 \wedge K}$$

Third constraint!

Constraint generation for abstraction:

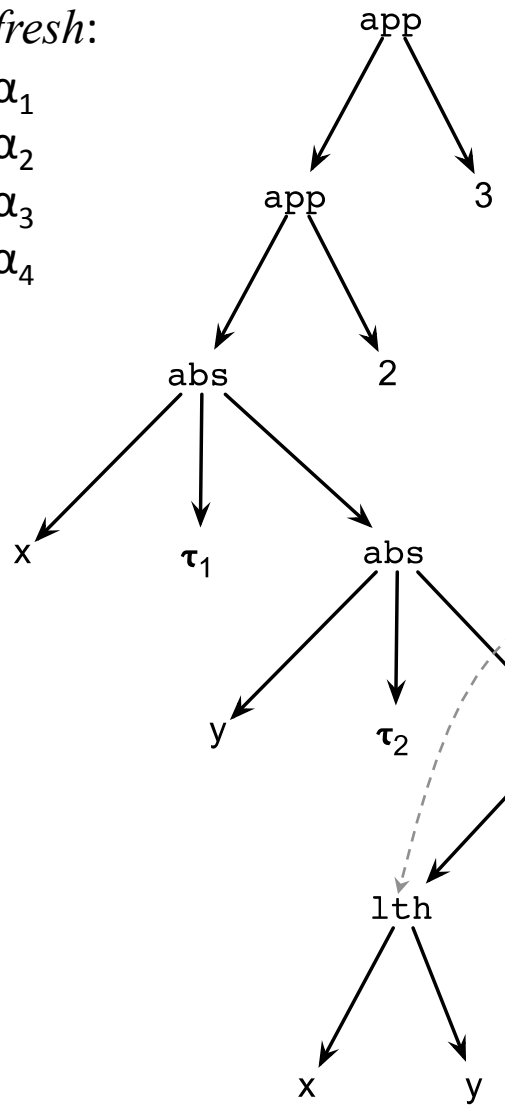
$$\frac{\text{fresh}(\alpha) \qquad \text{gen}(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{\text{gen}(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K}$$

gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)



gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)

fresh:
α_1
α_2
α_3
α_4

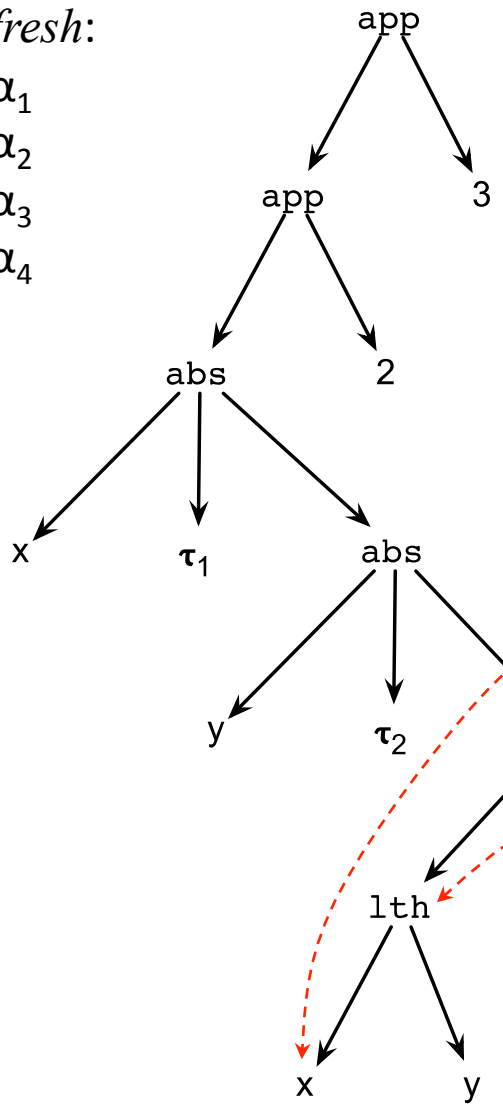


$$\frac{\text{gen}(\Gamma, E_1, \text{bool}) = K_1 \qquad \text{gen}(\Gamma, E_2, \tau) = K_2 \qquad \text{gen}(\Gamma, E_3, \tau) = K_3}{\text{gen}(\Gamma, \text{if } E_1 \text{ then } E_2 \text{ else } E_3, \tau) = K_1 \wedge K_2 \wedge K_3}$$

$$\frac{\begin{array}{l}\text{gen}(\{y \mapsto \tau_2, x \mapsto \tau_1\}, \text{lth}, \text{bool}) = K_1 \\ \text{gen}(\{y \mapsto \tau_2, x \mapsto \tau_1\}, x, \alpha_4) = K_2 \\ \text{gen}(\{y \mapsto \tau_2, x \mapsto \tau_1\}, y, \alpha_4) = K_3\end{array}}{\text{gen}(\{x \mapsto \tau_x\}, \text{if } x<y \text{ then } x \text{ else } y, \alpha_4) = K_1 \wedge K_2 \wedge K_3}$$

gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)

fresh:
α_1
α_2
α_3
α_4

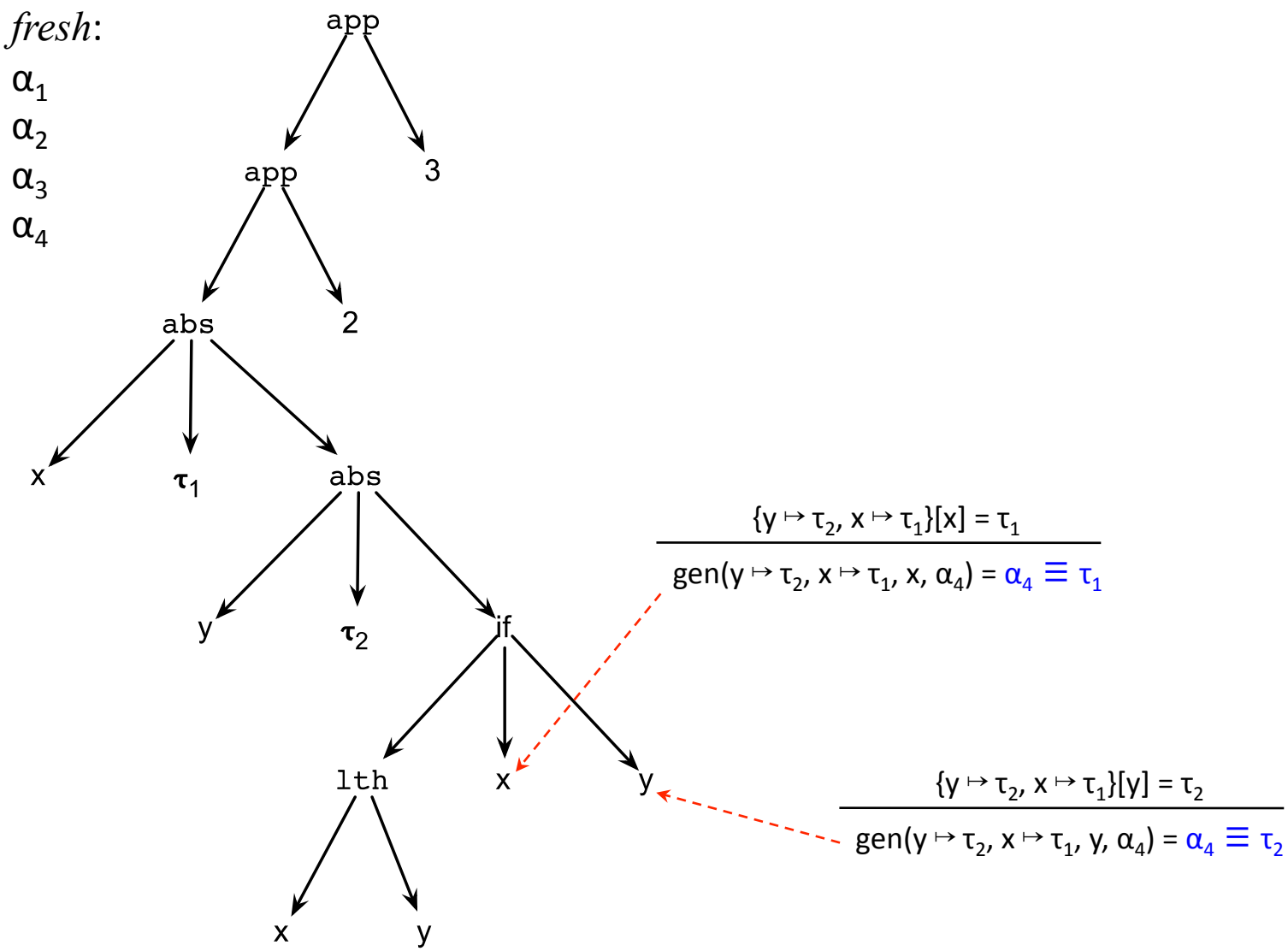


$$\frac{gen(\Gamma, E_1, int) = K_1 \qquad gen(\Gamma, E_2, int) = K_2}{gen(\Gamma, E_1<E_2, \tau) = \tau \equiv bool \wedge K_1 \wedge K_2}$$

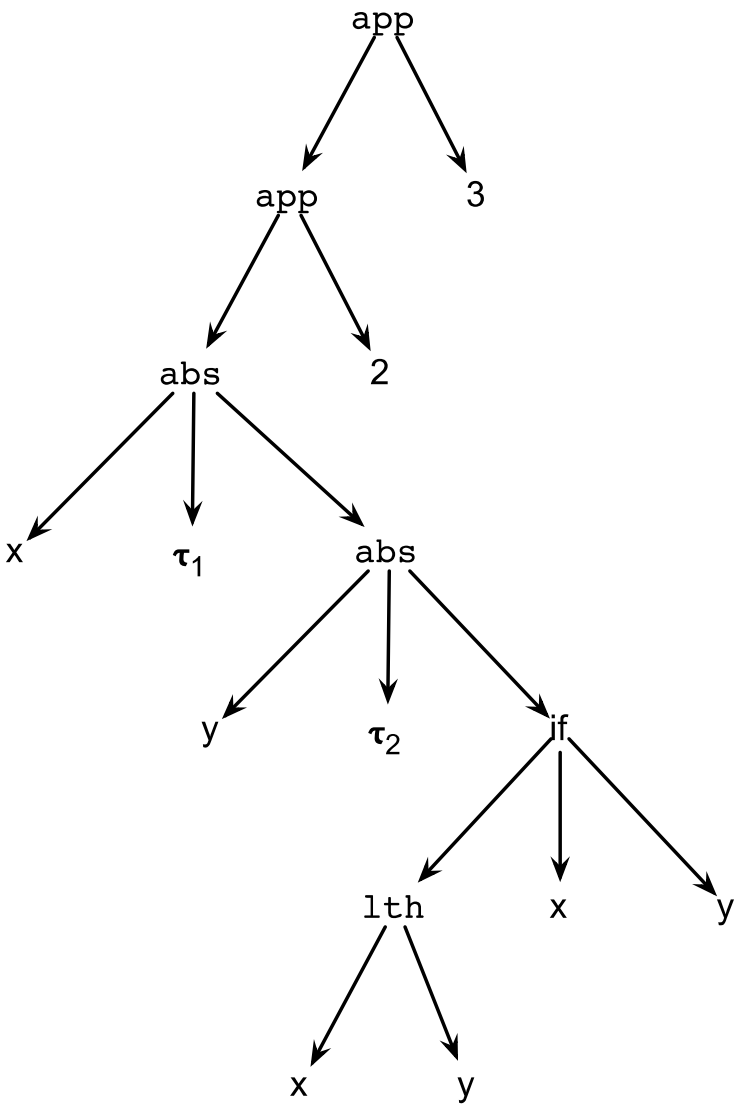
$$\frac{\Gamma[x] = \tau_x}{gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

$$\frac{\dfrac{\{y \mapsto \tau_2, x \mapsto \tau_1\}[x] = \tau_1}{gen(y \mapsto \tau_2, x \mapsto \tau_1, x, int) = int \equiv \tau_1} \qquad \dfrac{\{y \mapsto \tau_2, x \mapsto \tau_1\}[y] = \tau_2}{gen(y \mapsto \tau_2, x \mapsto \tau_1, y_2, int) = int \equiv \tau_2}}{gen(y \mapsto \tau_2, x \mapsto \tau_1, x<y, bool) = bool \equiv bool \wedge K_1 \wedge K_2}$$

gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)



gen(Ø, ((λx:τ_1.(λy:τ_2.(if x<y then x else y)))2)3, τ_0)



The full list of constraints:

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
bool ≡ bool ∧
int $\equiv \tau_1 \wedge$
int $\equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv$ int ∧
$\alpha_1 \equiv$ int

1. Can you solve these constraints to find the type of the whole program?
2. Can you think about a general way to solve these constraints?

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

lac.dcc.ufmg.br

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

UNIFICATION

fernando@dcc.ufmg.br

Constraints are solved via Unification

- A type variable unifies with itself, or with anything that does not contain itself
- Two ground types unify if they are the same (modulo alpha renaming of quantified variables)
- Two composite types unify if their subparts unify

What is alpha renaming?

The "Occur" Check

- A type variable unifies with itself, or with anything that does not contain itself
- Two ground types unify if they are the same (modulo alpha renaming of quantified variables)
- Two composite types unify if their subparts unify

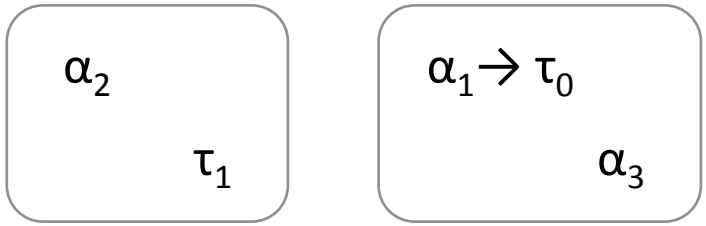
There is no way to solve "$\alpha_1 \equiv \tau_2 \rightarrow \alpha_1$". We would need an infinite sequence of α_1, e.g.: $\alpha_1 \equiv \tau_2 \rightarrow (\tau_2 \rightarrow (\tau_2 \rightarrow (\tau_2 \rightarrow (\tau_2 \rightarrow \ldots))))$

Example of Unification

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

$Unify(\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0), \tau_1 \rightarrow \alpha_3) \Rightarrow$

$Unify(\alpha_2, \tau_1)$ and $Unify(\alpha_1 \rightarrow \tau_0, \alpha_3)$

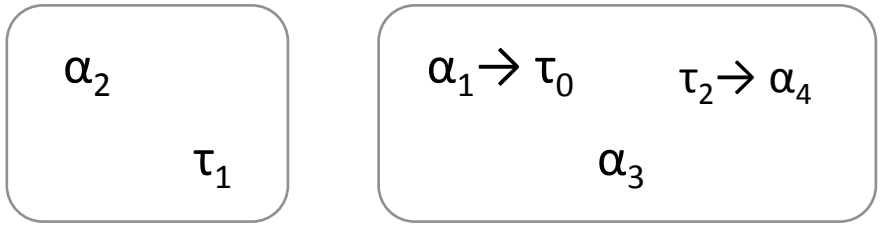


Unification groups types into equivalence classes

Adding more Constraints

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

Unify(α_3, $\tau_2 \rightarrow \alpha_4$)

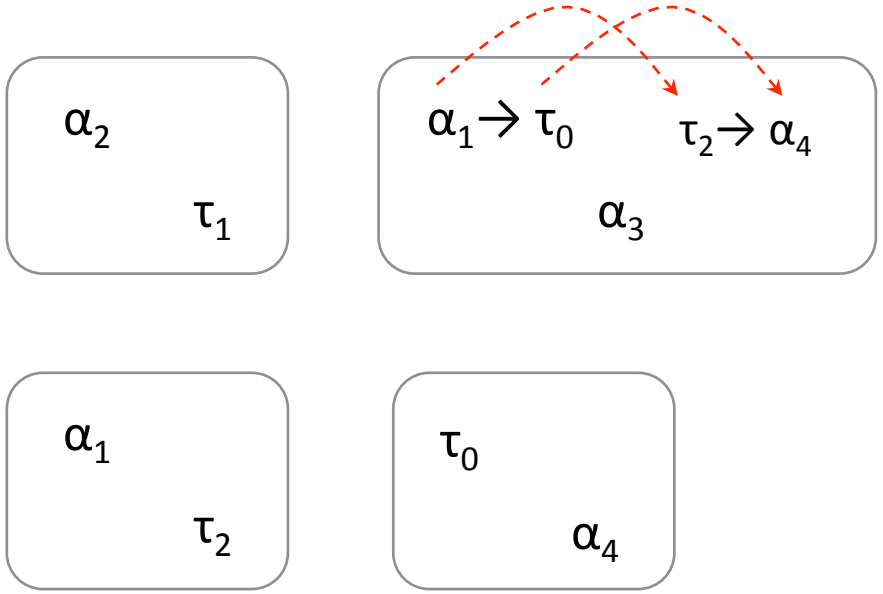


Unification groups types into equivalence classes

Adding more Constraints

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

Unify(α_3, $\tau_2 \rightarrow \alpha_4$)

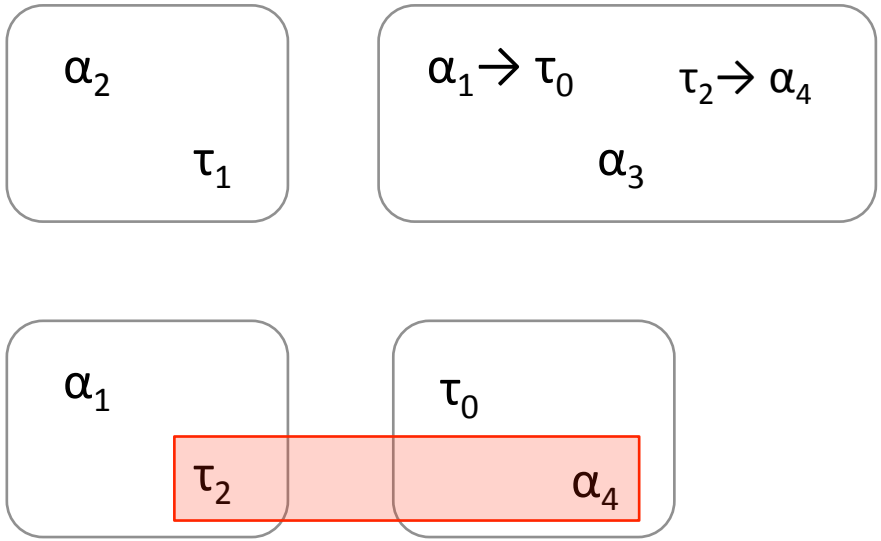


Which can be further refined

Some constraints unify whole eqv. classes

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

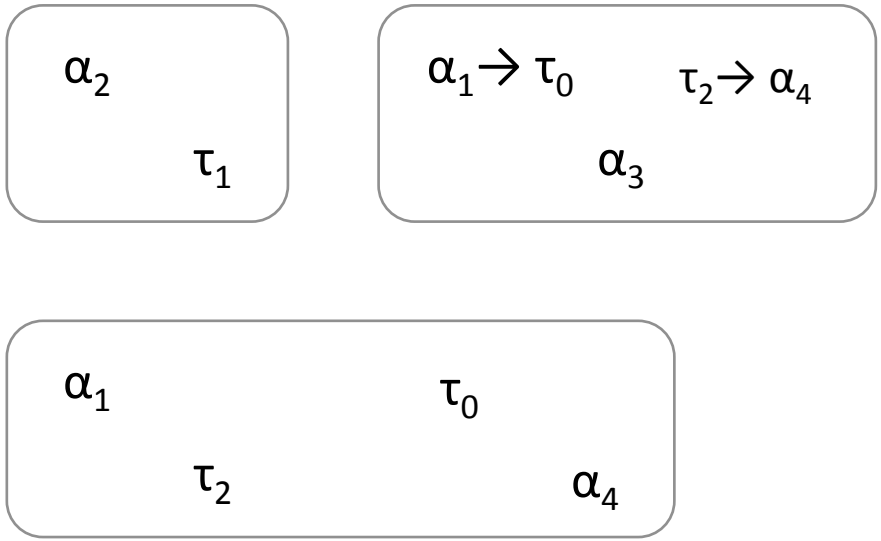
Unify(α_4, τ_2)



And others are no-ops

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
bool $\equiv$ bool $\wedge$
int $\equiv \tau_1 \wedge$
int $\equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv$ int $\wedge$
$\alpha_1 \equiv$ int

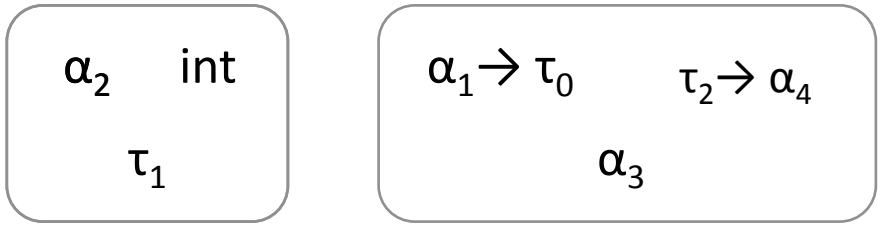
Unify(bool, bool)

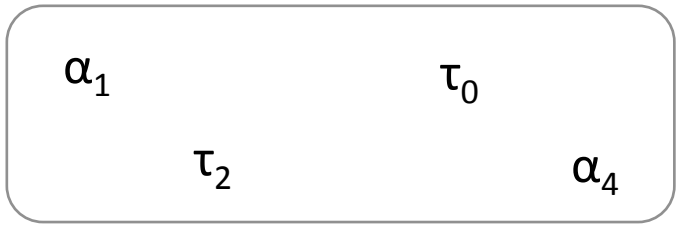


We go on...

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

Unify(int, τ_1)





We go on...

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

Unify(int, τ_2)

α_2 int

τ_1

$\alpha_1 \rightarrow \tau_0$ $\tau_2 \rightarrow \alpha_4$

α_3

α_1 int τ_0

τ_2 α_4

And on...

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

Unify(α_4, τ_1)

α_2	τ_1
τ_0	α_4
α_1	int
	τ_2

$\alpha_1 \rightarrow \tau_0$	$\tau_2 \rightarrow \alpha_4$
α_3	

Some constraints can simply be discarded

$\alpha_2 \rightarrow (\alpha_1 \rightarrow \tau_0) \equiv \tau_1 \rightarrow \alpha_3 \wedge$
$\alpha_3 \equiv \tau_2 \rightarrow \alpha_4 \wedge$
$\alpha_4 \equiv \tau_2 \wedge$
$bool \equiv bool \wedge$
$int \equiv \tau_1 \wedge$
$int \equiv \tau_2 \wedge$
$\alpha_4 \equiv \tau_1 \wedge$
$\alpha_2 \equiv int \wedge$
$\alpha_1 \equiv int$

Unify(α_2, int) and Unify(α_1, int)

α_2	τ_1
τ_0	α_4
α_1	int
	τ_2

$\alpha_1 \rightarrow \tau_0$ $\quad$ $\tau_2 \rightarrow \alpha_4$

α_3

A Declarative Solver

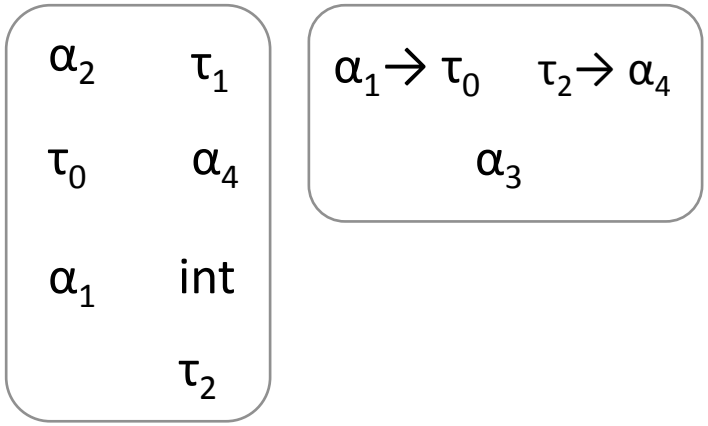
$$\text{solve}(\sigma, \emptyset) = \sigma$$

$$\frac{\text{solve}(\sigma, K) = \sigma'}{\text{solve}(\sigma, X \equiv Y \wedge K) = \sigma' \cup \text{Unify}(X, Y)}$$

$$\frac{\text{solve}(\sigma, K) = \sigma'}{\text{solve}(\sigma, X \equiv X \wedge K) = \sigma'}$$

$$\frac{\text{solve}(\sigma, W \equiv Y \wedge X \equiv Z \wedge K) = \sigma'}{\text{solve}(\sigma, W \to X \equiv Y \to Z \wedge K) = \sigma'}$$

σ is the result of the solver: a partition of the universe of type variables into equivalence classes.



Rewriting the Program

$((\lambda x{:}\tau_1.(\lambda y{:}\tau_2.(\text{if } x<y \text{ then } x \text{ else } y)))2)3,\ \tau_0$

Can you find suitable substitutions for τ_0, τ_1 and τ_2 so that the program above type-checks?

$\alpha_2 \quad \tau_1$

$\tau_0 \quad \alpha_4$

$\alpha_1 \quad \text{int}$

τ_2

$\alpha_1 \rightarrow \tau_0 \quad \tau_2 \rightarrow \alpha_4$

α_3

Finding names for Classes

$$\frac{int \in Q}{name(Q) = int} \qquad \frac{bool \in Q}{name(Q) = bool} \qquad \frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{name(Q) = name(Q_1) \to name(Q_2)}$$

$$\frac{int \notin Q \qquad bool \notin Q \qquad X \to Y \notin Q}{name(Q) = fresh\_type\_name}$$

Finding names for Classes

$$\frac{\text{int} \in Q}{\text{name}(Q) = \text{int}} \qquad \frac{\text{bool} \in Q}{\text{name}(Q) = \text{bool}} \qquad \frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{\text{name}(Q) = \text{name}(Q_1) \to \text{name}(Q_2)}$$

$$\frac{\text{int} \notin Q \qquad \text{bool} \notin Q \qquad X \to Y \notin Q}{\text{name}(Q) = \text{fresh\_type\_name}}$$

Whenever we generate fresh type names, we will end up with polymorphic types, e.g.: $\forall \tau.\tau \to \text{bool}$

Finding names for Classes

$$\frac{int \in Q}{name(Q) = int} \qquad \frac{bool \in Q}{name(Q) = bool} \qquad \frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{name(Q) = name(Q_1) \to name(Q_2)}$$

$$\frac{int \notin Q \qquad bool \notin Q \qquad X \to Y \notin Q}{name(Q) = fresh\_type\_name}$$

α_2 τ_1

τ_0 α_4

α_1 int

τ_2

What's the name of these classes?

$\alpha_1 \to \tau_0$ $\tau_2 \to \alpha_4$

α_3

Finding names for Classes

$$\frac{int \in Q}{name(Q) = int}$$

$$\frac{bool \in Q}{name(Q) = bool}$$

$$\frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{name(Q) = name(Q_1) \to name(Q_2)}$$

$$\frac{int \notin Q \qquad bool \notin Q \qquad X \to Y \notin Q}{name(Q) = fresh\_type\_name}$$

How to rewrite a program?

int

α_2 τ_1

τ_0 α_4

α_1 int

τ_2

$\alpha_1 \to \tau_0$ $\tau_2 \to \alpha_4$

α_3

Finding names for Classes

$$\frac{int \in Q}{name(Q) = int} \qquad \frac{bool \in Q}{name(Q) = bool}$$

$$\frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{name(Q) = name(Q_1) \to name(Q_2)}$$

$$\frac{int \notin Q \qquad bool \notin Q \qquad X \to Y \notin Q}{name(Q) = fresh\_type\_name}$$

How to rewrite a program?

int

α_2 τ_1

τ_0 α_4

α_1 int

τ_2

int→int

$\alpha_1 \to \tau_0$ $\tau_2 \to \alpha_4$

α_3

Rewriting programs: the rw function

What's the only rule that changes something?

$$rw(false) = false \qquad rw(true) = true$$

$$\frac{X \text{ is variable}}{rw(X) = X} \qquad \frac{c \text{ is integer literal}}{rw(c) = c}$$

$$\frac{rw(\sigma, E_1) = E_{11} \qquad rw(\sigma, E_2) = E_{22}}{rw(E_1 + E_2) = E_{11} + E_{22}}$$

$$\frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22} \qquad rw(E_3) = E_{33}}{rw(\text{if } E_1 \text{ then } E_2 \text{ else } E_3) = \text{if } E_{11} \text{ then } E_{22} \text{ else } E_{33}}$$

$$\frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22}}{rw(E_1 < E_2) = E_{11} < E_{22}} \qquad \frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22}}{rw(E_1 \text{ and } E_2) = E_{11} \text{ and } E_{22}}$$

$$\frac{\tau_x \in Q \qquad name(Q) = \tau_d \qquad rw(\sigma, E) = E'}{rw(\lambda x{:}\tau_x.E) = \lambda x{:}\tau_d.E'} \qquad \frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22}}{rw(E_1E_2) = E_{11}E_{22}}$$

Rewriting programs: the rw function

What's the only rule that changes something?

$$rw(false) = false \qquad rw(true) = true$$

$$\frac{X \text{ is variable}}{rw(X) = X} \qquad \frac{c \text{ is integer literal}}{rw(c) = c}$$

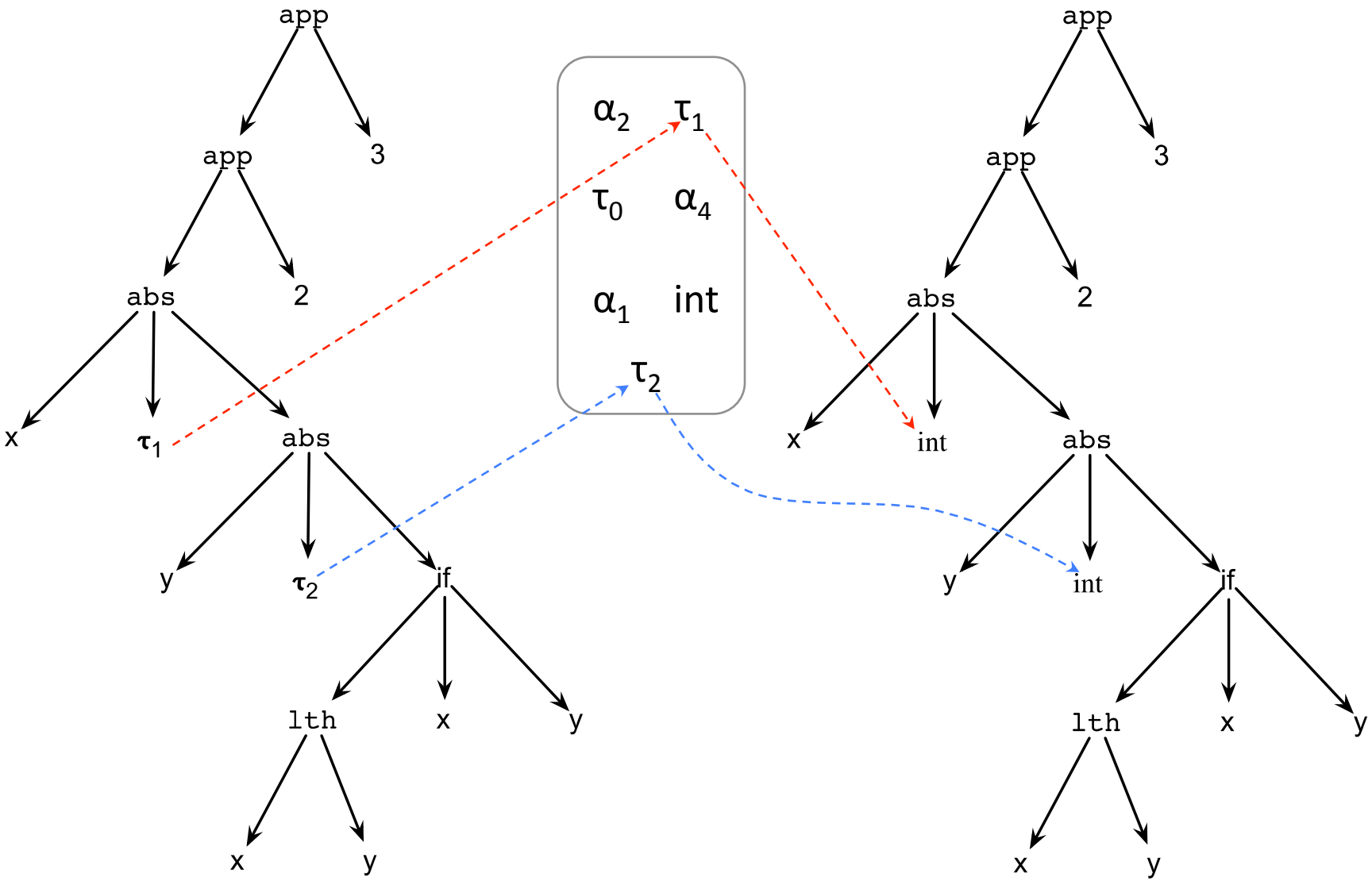
$$\frac{rw(\sigma, E_1) = E_{11} \qquad rw(\sigma, E_2) = E_{22}}{rw(E_1 + E_2) = E_{11} + E_{22}}$$

$$\frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22} \qquad rw(E_3) = E_{33}}{rw(\text{if } E_1 \text{ then } E_2 \text{ else } E_3) = \text{if } E_{11} \text{ then } E_{22} \text{ else } E_{33}}$$

$$\frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22}}{rw(E_1 < E_2) = E_{11} < E_{22}} \qquad \frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22}}{rw(E_1 \text{ and } E_2) = E_{11} \text{ and } E_{22}}$$

$$\frac{\tau_x \in Q \qquad name(Q) = \tau_d \qquad rw(\sigma, E) = E'}{rw(\lambda x{:}\tau_x.E) = \lambda x{:}\tau_d.E'} \qquad \frac{rw(E_1) = E_{11} \qquad rw(E_2) = E_{22}}{rw(E_1E_2) = E_{11}E_{22}}$$

Example of Rewriting Pass



A Complete Example

$\lambda w{:}\tau_4.\ \lambda x{:}\tau_3.\ \lambda y{:}\tau_2.\ \lambda z{:}\tau_1.$ if w > x then y else z y

Generate constraints for the program above.

A Complete Example

$\lambda w{:}\tau_4.\ \lambda x{:}\tau_3.\ \lambda y{:}\tau_2.\ \lambda z{:}\tau_1.$ if w > x then y else z y

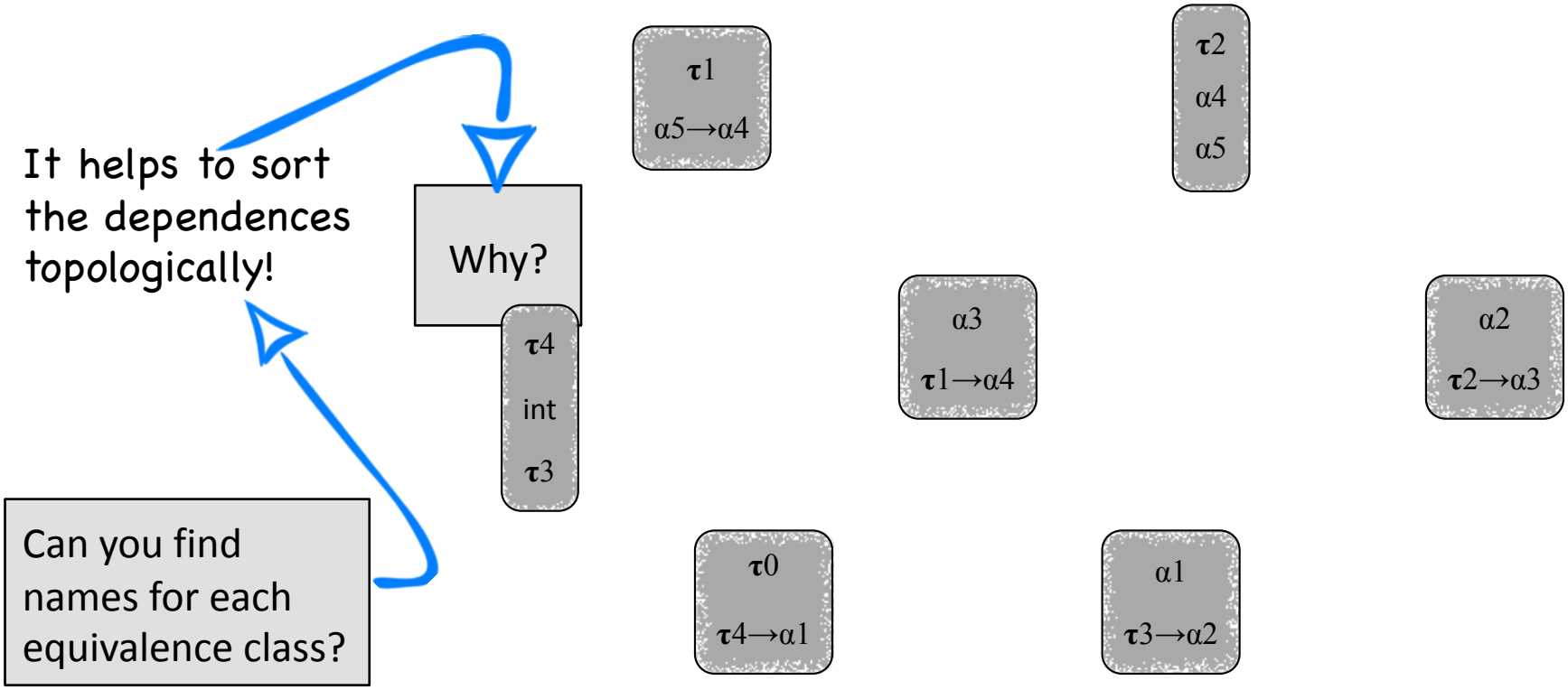
$\tau_0 \equiv \tau_4 \to \alpha_1 \wedge \alpha_1 \equiv \tau_3 \to \alpha_2 \wedge \alpha_2 \equiv \tau_2 \to \alpha_3 \wedge \alpha_3 \equiv \tau_1 \to \alpha_4 \wedge \alpha_5 \to \alpha_4 \equiv \tau_1 \wedge \alpha_5 \equiv \tau_2 \wedge \text{bool} \equiv \text{bool} \wedge \tau_4 \equiv \text{int} \wedge \tau_3 \equiv \text{int} \wedge \alpha_4 \equiv \tau_2$

Can you solve these constraints?

A Complete Example

$\lambda w{:}\tau_4.\ \lambda x{:}\tau_3.\ \lambda y{:}\tau_2.\ \lambda z{:}\tau_1.$ if w > x then y else z y

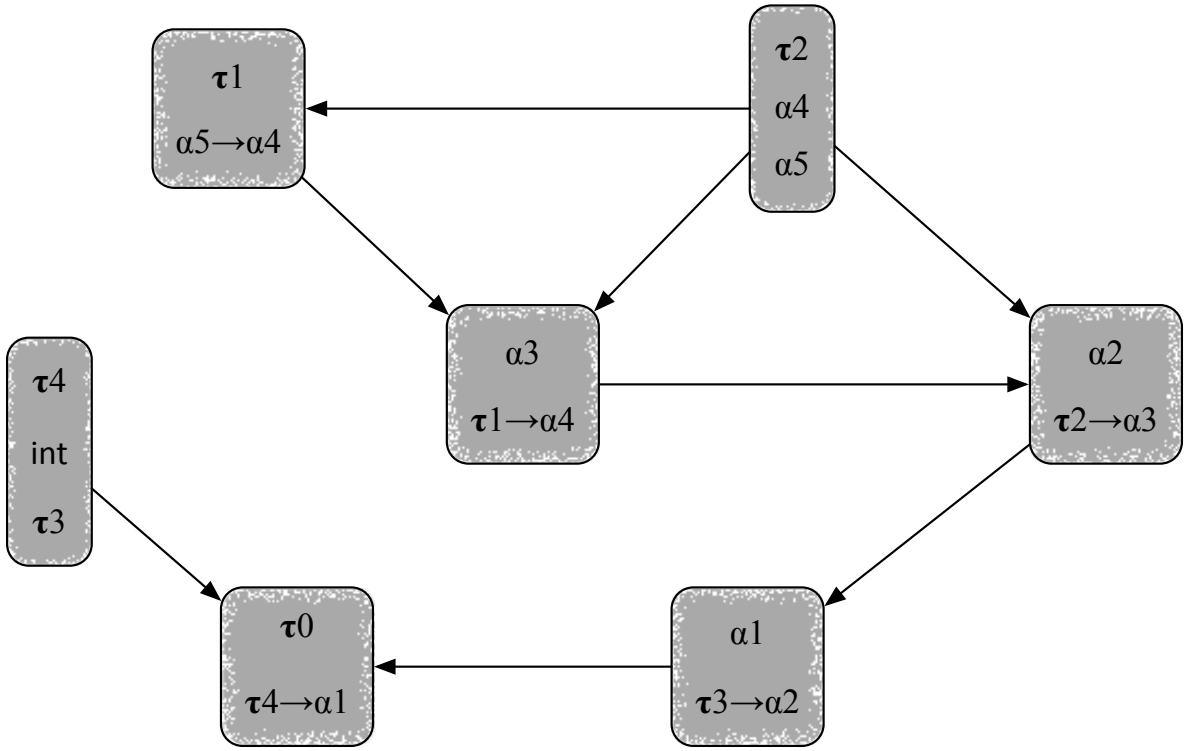
$\tau_0 \equiv \tau_4 \to \alpha_1 \wedge \alpha_1 \equiv \tau_3 \to \alpha_2 \wedge \alpha_2 \equiv \tau_2 \to \alpha_3 \wedge \alpha_3 \equiv \tau_1 \to \alpha_4 \wedge \alpha_5 \to \alpha_4 \equiv \tau_1 \wedge \alpha_5 \equiv \tau_2 \wedge \text{bool} \equiv \text{bool} \wedge \tau_4 \equiv \text{int} \wedge \tau_3 \equiv \text{int} \wedge \alpha_4 \equiv \tau_2$



A Complete Example

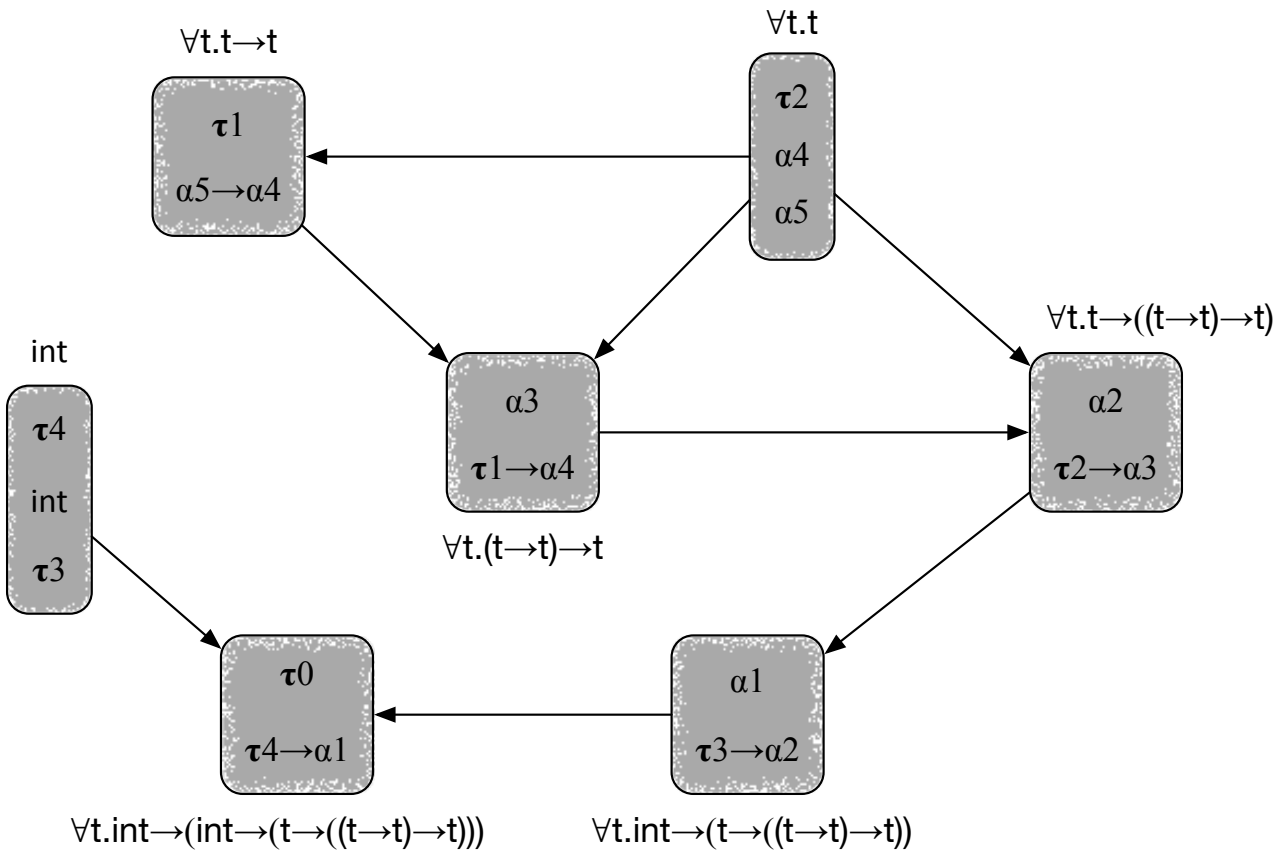
$\lambda w{:}\tau_4.\ \lambda x{:}\tau_3.\ \lambda y{:}\tau_2.\ \lambda z{:}\tau_1.$ if w > x then y else z y

$\tau_0 \equiv \tau_4 \to \alpha_1 \wedge \alpha_1 \equiv \tau_3 \to \alpha_2 \wedge \alpha_2 \equiv \tau_2 \to \alpha_3 \wedge \alpha_3 \equiv \tau_1 \to \alpha_4 \wedge \alpha_5 \to \alpha_4 \equiv \tau_1 \wedge \alpha_5 \equiv \tau_2 \wedge \text{bool} \equiv \text{bool} \wedge \tau_4 \equiv \text{int} \wedge \tau_3 \equiv \text{int} \wedge \alpha_4 \equiv \tau_2$



So, now can you find names for each class?

A Complete Example

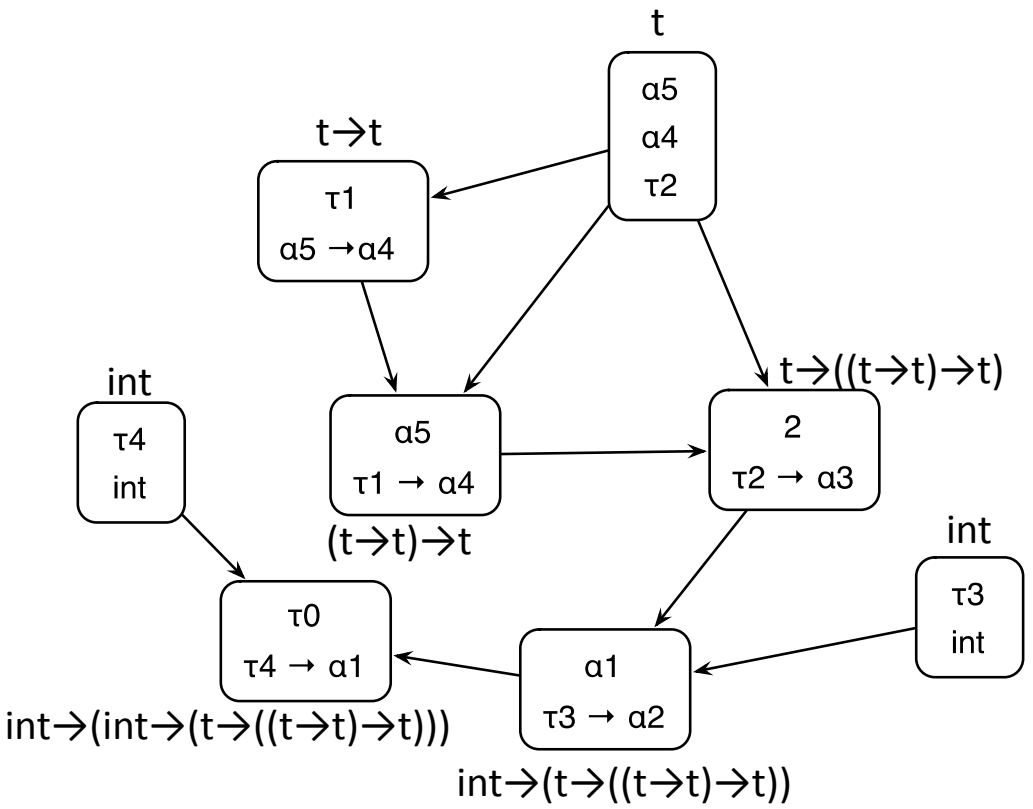


Now, can you rewrite the program?

A Complete Example

$\lambda w{:}\tau_4.\ \lambda x{:}\tau_3.\ \lambda y{:}\tau_2.\ \lambda z{:}\tau_1.$ if w > x then y else z y

$\forall t.\lambda w{:}int.\ \lambda x{:}int.\ \lambda y{:}t.\ \lambda z{:}t\rightarrow t.$ if w > x then y else z y



A Bit of History

- In this class we use the algorithm of Melo *et al*., which is based on the approach of Pottier and Rémy.
- The classic algorithm of type inference is due to Hindley, Damas and Milner.
- Unification was invented by Allan Robinson.
- Chapter 22 in *Types and Programming Languages* talks about type inference.

- François Pottier and Didier Rémy, *The Essence of ML Type Inference*, (2003)
- L. Melo, R. Ribeiro, M. Araújo, F. Pereira: *Inference of static semantics for incomplete C programs*. PACMPL 2(POPL): 29:1-29:28 (2018)
- Robinson, J. Alan. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM. 12 (1): 23–41, (1965)
- Damas, Luis; Milner, Robin (1982), *Principal type-schemes for functional programs*, POPL, ACM, pp. 207–212, (1982)

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

CORRECTNESS

The Essential Theorem

Given

- Expression E
- Typing environment Γ mapping variables to ground types
- Substitution σ

if

1. For all V ∈ fv(E), V ∈ domain(Γ)
2. gen(Γ, E, τ) = K
3. solve(σ, K) = σ'
4. rw(σ", E) = E', for some σ" ≥ σ'

then

- $\Gamma \vdash$ E':σ"(τ)

What does it mean σ" ≥ σ'?

$\sigma'' \geq \sigma'$

- Given substitutions σ' and σ'', we say that $\sigma'' \geq \sigma'$ if $\sigma''[\tau_1] = \sigma''[\tau_2] \Rightarrow \sigma'[\tau_1] = \sigma'[\tau_2]$

Lemma 1: $solve(\sigma, K) = \sigma' \Rightarrow \sigma' \geq \sigma$

Prove it.

Where:

1. $solve(\sigma, \emptyset) = \sigma$
2. $solve(\sigma, X \equiv X \wedge K) = solve(\sigma, K)$
3. $solve(\sigma, W \rightarrow X \equiv Y \rightarrow Z \wedge K) = solve(\sigma, W \equiv Y \wedge X \equiv Z \wedge K)$
4. $solve(\sigma, X \equiv Y \wedge K) = \text{if } X \notin fv(Y) \text{ then } solve(\sigma \cup Unify(X, Y), K) \text{ else error}$

Lemma 1: solve(σ, K) = σ' ⇒ σ' ≥ σ

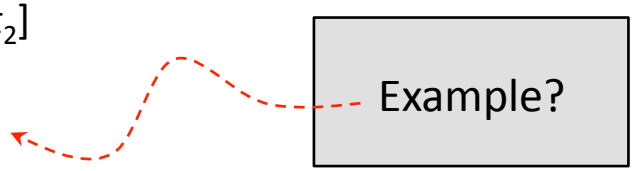
1. solve(σ, ∅) = σ
2. solve(σ, X ≡ X ∧ K) = solve(σ, K)
3. solve(σ, W→X ≡ Y→Z ∧ K) = solve(σ, W ≡ Y ∧ X ≡ Z ∧ K)
4. solve(σ, X ≡ Y ∧ K) = if X ∉ fv(Y) then solve(σ ∪ Unify(X, Y), K) else error

Proof: induction on the number of invocations of solve (on the height of solve's derivation tree). The first three rules are trivial, because they do not change σ. The last rule (4) either adds a new equivalence class to σ, or unifies two equivalence classes already there. In any case, the key equality is preserved:

$\sigma[\tau_1] = \sigma[\tau_2] \Rightarrow (\sigma \cup \text{Unify}(X, Y))[\tau_1] = (\sigma \cup \text{Unify}(X, Y))[\tau_2]$

Notice that the contrary might not be true, e.g.:

$(\sigma \cup \text{Unify}(X, Y))[\tau_1] = (\sigma \cup \text{Unify}(X, Y))[\tau_2] \not\Rightarrow \sigma[\tau_1] = \sigma[\tau_2]$



Lemma 2: name(Q) is a Ground Type

$$\frac{\text{int} \in Q}{\text{name}(Q) = \text{int}}$$

$$\frac{\text{bool} \in Q}{\text{name}(Q) = \text{bool}}$$

$$\frac{\text{int} \notin Q \qquad \text{bool} \notin Q \qquad X \to Y \notin Q}{\text{name}(Q) = \text{fresh\_type\_name}}$$

$$\frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{\text{name}(Q) = \text{name}(Q_1) \to \text{name}(Q_2)}$$

Prove it.

Lemma 2: name(Q) is a Ground Type

$$\frac{\text{int} \in Q}{\text{name}(Q) = \text{int}}$$

int is a ground type

$$\frac{\text{bool} \in Q}{\text{name}(Q) = \text{bool}}$$

bool is a ground type

$$\frac{\text{int} \notin Q \qquad \text{bool} \notin Q \qquad X \to Y \notin Q}{\text{name}(Q) = \text{fresh\_type\_name}}$$

$\forall$ t.t is a ground type

$$\frac{X \to Y \in Q \qquad X \in Q_1 \qquad Y \in Q_2}{\text{name}(Q) = \text{name}(Q_1) \to \text{name}(Q_2)}$$

Induction on Q_1 and on Q_2 plus the fact that if t and s are ground types, so is t→s

Back to the Key Theorem

Given

- Expression E
- Typing environment Γ[§]
- Substitution σ

if

1. For all V ∈ fv(E), V ∈ domain(Γ)
2. gen(Γ, E, τ) = K
3. solve(σ, K) = σ'
4. rw(σ'', E) = E', for some σ'' ≥ σ'

then

- $\Gamma \vdash E' : \sigma''(\tau)$

1. What kind of induction can we use to prove it?
2. Have you seen something similar before?

[§]: the environment maps variables to ground types

Back to the Key Theorem

Given

- Expression E
- Typing environment Γ
- Substitution σ

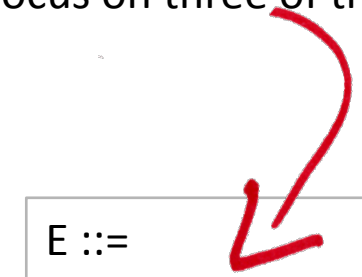
if

1. For all V ∈ fv(E), V ∈ domain(Γ)
2. gen(Γ, E, τ) = K
3. solve(σ, K) = σ'
4. rw(σ", E) = E', for some σ" ≥ σ'

then

- $\Gamma \vdash E' : \sigma''(\tau)$

We shall do case analysis on the expressions. There are eight cases to consider. We shall focus on three of them.



E ::=
- x
- λx:T.E
- E E
- *c*
- E + E
- E < E
- E and E
- if E then E else E

Expression is "λx:T.E"

Given
- Expression λx:T.E
- Typing environment Γ
- Substitution σ

if

1. For all $V \in fv(\lambda x{:}\tau_x.E)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K$
3. $solve(\sigma, \tau \equiv \tau_x \to \alpha \wedge K) = \sigma' \cup Unify(\tau, \tau_x \to \alpha)$
4. let $\sigma'' = \sigma \cup Unify(\tau, \tau_x \to \alpha)$ in $rw(\sigma'', \lambda x{:}\tau_x.E) = \lambda x{:}\sigma''(\tau_d).E'$

then
- $\Gamma \vdash \lambda x{:}\sigma''(\tau_x).E' : \sigma''(\tau_x) \to \sigma''(\alpha)$

$fv(\lambda x{:}T.E) = fv(E) - \{x\}$
For all $V \in fv(E) - \{x\}$, $V \in domain(\Gamma)$
For all $V \in fv(E)$, $V \in domain(\Gamma \cup \{x \mapsto \tau_x\})$

$$\frac{fresh(\alpha) \qquad gen(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{gen(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K}$$

$$\frac{solve(\sigma, K) = \sigma'}{solve(\sigma, \tau \equiv \tau_x \to \alpha \wedge K) = \sigma' \cup Unify(\tau, \tau_x \to \alpha)}$$

$$\frac{\tau_x \in Q \qquad name(Q) = \tau_d \qquad rw(\sigma'', E) = E'}{rw(\sigma'', \lambda x{:}\tau_x.E) = \lambda x{:}\tau_d.E'}$$

You need 4 facts to apply induction. Can you spot them?

Induction on E, taken from λx:T.E

Given
- Expression E
- Typing environment $\Gamma \cup \{x \mapsto \tau_x\}$
- Substitution σ

if
1. For all $V \in fv(E)$, $V \in domain(\Gamma \cup \{x \mapsto \tau_x\})$
2. $gen(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K$
3. $solve(\sigma, K) = \sigma'$
4. let $\sigma'' = \sigma \cup Unify(\tau, \tau_x \to \alpha)$ in $rw(\sigma'', E) = E'$

then
- $\Gamma \cup \{x \mapsto \tau_x\} \vdash E' : \sigma''(\alpha)$

$fv(\lambda x{:}T.E) = fv(E) - \{x\}$
For all $V \in fv(E) - \{x\}$, $V \in domain(\Gamma)$
For all $V \in fv(E)$, $V \in domain(\Gamma \cup \{x \mapsto \tau_x\})$

$$\frac{fresh(\alpha) \qquad gen(\Gamma \cup \{x \mapsto \tau_x\}, E, \alpha) = K}{gen(\Gamma, \lambda x{:}\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K}$$

$$\frac{solve(\sigma, K) = \sigma'}{solve(\sigma, \tau \equiv \tau_x \to \alpha \wedge K) = \sigma' \cup Unify(\tau, \tau_x \to \alpha)}$$

$$\frac{\tau_x \in Q \qquad name(Q) = \tau_d \qquad rw(\sigma'', E) = E'}{rw(\sigma'', \lambda x{:}\tau_x.E) = \lambda x{:}\tau_d.E'}$$

What can we do, now that we know the fact below?
$\Gamma \cup \{x \mapsto \tau_x\} \vdash E' : \sigma''(\alpha)$

Invoking T-ABS

$$\frac{\Gamma \cup \{x \mapsto T_1\} \vdash E:T_2}{\Gamma \vdash (\lambda x:T_1.E): T_1 \to T_2} \quad [\text{T-ABS}]$$

Given

- Expression $\lambda x:T.E$
- Typing environment Γ
- Substitution σ

if

1. For all $V \in fv(\lambda x:\tau_x.E)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, \lambda x:\tau_x.E, \tau) = \tau \equiv \tau_x \to \alpha \wedge K$
3. $solve(\sigma, \tau \equiv \tau_x \to \alpha \wedge K) = \sigma' \cup Unify(\tau, \tau_x \to \alpha)$
4. let $\sigma'' = \sigma \cup Unify(\tau, \tau_x \to \alpha)$ in $rw(\sigma'', \lambda x:\tau_x.E) = \lambda x:\sigma''(\tau_d).E'$

then

- $\Gamma \vdash \lambda x:\sigma''(\tau_x).E':\sigma''(\tau_x) \to \sigma''(\alpha)$

$\Rightarrow \Gamma \vdash \lambda x:\sigma''(\tau_x).E':\sigma''(\tau)$

How is this last implication possible?
E.g.:
$\sigma''(\tau_x) \to \sigma''(\alpha) = \sigma''(\tau)$

$$\frac{\Gamma \cup \{x \mapsto \tau_x\} \vdash E':\sigma''(\alpha)}{\Gamma \vdash \lambda x:\sigma''(\tau_x).E':\sigma''(\tau_x) \to \sigma''(\alpha)}$$

From induction:
$\Gamma \cup \{x \mapsto \tau_x\} \vdash E':\sigma''(\alpha)$

Expression is "E_1E_2"

Given
- Expression E_1E_2
- Typing environment Γ
- Substitution σ

if
1. For all $V \in fv(E_1) \cup fv(E_2)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, E_1E_2, \tau) = \tau \equiv K_1 \wedge K_2$
3. $solve(\sigma, K_1 \wedge K_2) = \sigma''$
4. $rw(\sigma'', E_1E_2) = E_{11}E_{22}$

then
- $\Gamma \vdash E_{11}E_{22} : \sigma''(\tau)$

This time, you will need to apply induction twice. Why?

$fv(E_1E_2) = fv(E_1) \cup fv(E_2)$
For all $V \in fv(E_1) \cup fv(E_2)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_1)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_2)$, $V \in domain(\Gamma)$

$$\frac{fresh(\alpha) \quad gen(\Gamma, E_1, \alpha \rightarrow \tau) = K_1 \quad gen(\Gamma, E_2, \alpha) = K_2}{gen(\Gamma, E_1E_2, \tau) = K_1 \wedge K_2}$$

$$\frac{solve(\sigma, K_1) = \sigma' \qquad solve(\sigma', K_2) = \sigma''}{solve(\sigma, K_1 \wedge K_2) = \sigma''}$$

$$\frac{rw(\sigma'', E_1) = E_{11} \qquad rw(\sigma'', E_2) = E_{22}}{rw(\sigma'', E_1E_2) = E_{11}E_{22}}$$

First induction: E_1

Given
- Expression E_1
- Typing environment Γ
- Substitution σ

if
1. For all $V \in fv(E_1)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, E_1, \alpha \rightarrow \tau) = K_1$
3. $solve(\sigma, K_1) = \sigma'$
4. $rw(\sigma'', E_1) = E_{11}$, $\sigma'' > \sigma'$

then
- $\Gamma \vdash E_{11} : \sigma''(\alpha \rightarrow \tau)$

How do we know that $\sigma'' > \sigma'$?

$fv(E_1E_2) = fv(E_1) \cup fv(E_2)$
For all $V \in fv(E_1) \cup fv(E_2)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_1)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_2)$, $V \in domain(\Gamma)$

$$\frac{fresh(\alpha) \quad gen(\Gamma, E_1, \alpha \rightarrow \tau) = K_1 \quad gen(\Gamma, E_2, \alpha) = K_2}{gen(\Gamma, E_1E_2, \tau) = K_1 \wedge K_2}$$

$$\frac{solve(\sigma, K_1) = \sigma' \qquad solve(\sigma', K_2) = \sigma''}{solve(\sigma, K_1 \wedge K_2) = \sigma''}$$

$$\frac{rw(\sigma'', E_1) = E_{11} \qquad rw(\sigma'', E_2) = E_{22}}{rw(\sigma'', E_1E_2) = E_{11}E_{22}}$$

First induction: E_1

Given
- Expression E_1
- Typing environment Γ
- Substitution σ

if
1. For all $V \in fv(E_1)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, E_1, \alpha \rightarrow \tau) = K_1$
3. $solve(\sigma, K_1) = \sigma'$
4. $rw(\sigma'', E_1) = E_{11}$, $\sigma'' > \sigma'$

then
- $\Gamma \vdash E_{11} : \sigma''(\alpha \rightarrow \tau)$

$fv(E_1E_2) = fv(E_1) \cup fv(E_2)$

For all $V \in fv(E_1) \cup fv(E_2)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_1)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_2)$, $V \in domain(\Gamma)$

$$\frac{fresh(\alpha) \quad gen(\Gamma, E_1, \alpha \rightarrow \tau) = K_1 \quad gen(\Gamma, E_2, \alpha) = K_2}{gen(\Gamma, E_1E_2, \tau) = K_1 \wedge K_2}$$

$$\frac{solve(\sigma, K_1) = \sigma' \qquad solve(\sigma', K_2) = \sigma''}{solve(\sigma, K_1 \wedge K_2) = \sigma''}$$

$$\frac{rw(\sigma'', E_1) = E_{11} \qquad rw(\sigma'', E_2) = E_{22}}{rw(\sigma'', E_1E_2) = E_{11}E_{22}}$$

How do we know that $\sigma'' > \sigma'$?

Second induction: E_2

Given
- Expression E_2
- Typing environment Γ
- Substitution σ'

if
1. For all $V \in fv(E_2)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, E_2, \alpha) = K_2$
3. $solve(\sigma', K_2) = \sigma''$
4. $rw(\sigma'', E_2)$

then
- $\Gamma \vdash E_{22}:\sigma''(\alpha)$

$fv(E_1E_2) = fv(E_1) \cup fv(E_2)$
For all $V \in fv(E_1) \cup fv(E_2)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_1)$, $V \in domain(\Gamma)$
- For all $V \in fv(E_2)$, $V \in domain(\Gamma)$

$$\frac{fresh(\alpha) \quad gen(\Gamma, E_1, \alpha \to \tau) = K_1 \quad gen(\Gamma, E_2, \alpha) = K_2}{gen(\Gamma, E_1E_2, \tau) = K_1 \wedge K_2}$$

$$\frac{solve(\sigma, K_1) = \sigma' \qquad solve(\sigma', K_2) = \sigma''}{solve(\sigma, K_1 \wedge K_2) = \sigma''}$$

$$\frac{rw(\sigma'', E_1) = E_{11} \qquad rw(\sigma'', E_2) = E_{22}}{rw(\sigma'', E_1E_2) = E_{11}E_{22}}$$

We now have:
1. $\Gamma \vdash E_{11}:\sigma''(\alpha \to \tau)$
2. $\Gamma \vdash E_{22}:\sigma''(\alpha)$

What can we conclude?

Invoking T-APP

Given
- Expression E_1E_2
- Typing environment Γ
- Substitution σ

if
1. For all $V \in fv(E_1) \cup fv(E_2)$, $V \in domain(\Gamma)$
2. $gen(\Gamma, E_1E_2, \tau) = \tau \equiv K_1 \wedge K_2$
3. $solve(\sigma, K_1 \wedge K_2) = \sigma''$
4. $rw(\sigma'', E_1E_2) = E_{11}E_{22}$

then
- $\Gamma \vdash E_{11}E_{22}:\sigma''(\tau)$

$$\frac{\Gamma \vdash E_1:T \to T_1 \qquad \Gamma \vdash E_2:T}{\Gamma \vdash E_1E_2: T_1} \quad [\text{T-APP}]$$

$$\frac{\Gamma \vdash E_{11}:\sigma''(\alpha \to \tau) \qquad \Gamma \vdash E_{22}:\sigma''(\alpha)}{\Gamma \vdash E_{11}E_{22}:\sigma''(\tau)}$$

We now have:
1. $\Gamma \vdash E_{11}:\sigma''(\alpha \to \tau)$
2. $\Gamma \vdash E_{22}:\sigma''(\alpha)$

What can we conclude?

Expression is a variable, e.g.: "x"

Given
- Expression x
- Typing environment Γ
- Substitution σ

if
1. $x \in domain(\Gamma)$
2. $gen(\Gamma, x, \tau) = \tau \equiv \tau_x$
3. $solve(\sigma, \tau \equiv \tau_x) = \sigma \cup Unify(\tau, \tau_x)$
4. $rw(\sigma \cup Unify(\tau, \tau_x), x) = x$

then
- $\Gamma \vdash x:(\sigma \cup Unify(\tau, \tau_x))(\tau)$

$fv(x) = \{x\}$
For all $V \in fv(x)$, $V \in domain(\Gamma)$
- $x \in domain(\Gamma)$

$$\frac{\Gamma[x] = \tau_x}{gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

$solve(\sigma, \tau \equiv \tau_x) = \sigma \cup Unify(\tau, \tau_x)$

$rw(\sigma \cup Unify(\tau, \tau_x), x) = x$

1. You will not need induction here. Why?
2. How do you prove this part then?

Building $\Gamma \vdash x:(\sigma \cup Unify(\tau, \tau_x))(\tau)$

Given

- Expression x
- Typing environment Γ
- Substitution σ

if

1. $x \in domain(\Gamma)$
2. $gen(\Gamma, x, \tau) = \tau \equiv \tau_x$
3. $solve(\sigma, \tau \equiv \tau_x) = \sigma \cup Unify(\tau, \tau_x)$
4. $rw(\sigma \cup Unify(\tau, \tau_x), x) = x$

then

- $\Gamma \vdash x:(\sigma \cup Unify(\tau, \tau_x))(\tau)$
- $\Gamma \vdash x:\tau_x$

$$\frac{\dfrac{\Gamma[x] = \tau_x}{x:\tau_x \in \Gamma}}{\Gamma \vdash x:\tau_x} \qquad \frac{\Gamma[x] = \tau_x}{gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

Why is this last part possible?

Building $\Gamma \vdash x:(\sigma \cup Unify(\tau, \tau_x))(\tau)$

Given
- Expression x
- Typing environment Γ
- Substitution σ

if
1. $x \in domain(\Gamma)$
2. $gen(\Gamma, x, \tau) = \tau \equiv \tau_x$
3. $solve(\sigma, \tau \equiv \tau_x) = \sigma \cup Unify(\tau, \tau_x)$
4. $rw(\sigma \cup Unify(\tau, \tau_x), x) = x$

then
- $\Gamma \vdash x:(\sigma \cup Unify(\tau, \tau_x))(\tau)$
- $\Gamma \vdash x:\tau_x$

$$\frac{\Gamma[x] = \tau_x \qquad \Gamma[x] = \tau_x}{\dfrac{x:\tau_x \in \Gamma}{\Gamma \vdash x:\tau_x} \qquad gen(\Gamma, x, \tau) = \tau \equiv \tau_x}$$

Why is this last part possible?

From Lemma 2, $\sigma \cup Unify(\tau, \tau_x)$ maps types to ground types. By hypothesis, τ_x is a ground type, because $\Gamma[x] = \tau_x$ And from the naming system, we know that $\sigma \cup Unify(\tau, \tau_x)(x)$ yields a ground type.

A Bit of History

- In this class we use the algorithm of Melo *et al*., which is based on the approach of Pottier and Rémy.
- The classic algorithm of type inference is due to Hindley, Damas and Milner.
- Unification was invented by Allan Robinson.
- Chapter 22 in *Types and Programming Languages* talks about type inference.

- François Pottier and Didier Rémy, *The Essence of ML Type Inference*, (2003)
- L. Melo, R. Ribeiro, M. Araújo, F. Pereira: *Inference of static semantics for incomplete C programs*. PACMPL 2(POPL): 29:1-29:28 (2018)
- Robinson, J. Alan. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM. 12 (1): 23–41, (1965)
- Damas, Luis; Milner, Robin (1982), *Principal type-schemes for functional programs*, POPL, ACM, pp. 207–212, (1982)