



Functional Programming

Lecture 10

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, **infinite data structures**, input and output, lambda-calculus, **lazy evaluation**, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- Type Checking
- Unification and its Implementation
- Type Inference

Type Checking

Problem – Type Checking

input: expression e and type τ

output: YES (e has type τ) or NO

Problem – Type Checking

input: expression e and type τ

output: YES (e has type τ) or NO

Haskell Examples

```
(\x -> x) True :: Bool
```

Problem – Type Checking

input: expression e and type τ

output: YES (e has type τ) or NO

Haskell Examples

```
(\x -> x) True :: Bool
```

```
double :: Int -> Int
```

```
double x = x + x
```


The Language of Expressions – Core FP

$e ::=$	$x \mid e e \mid \lambda x. e$	λ -terms
	c	constant (for primitives)
	let $x = e$ in e	let-binding
	if e then e else e	conditional branching

The Language of Expressions – Core FP

$e ::= x \mid e e \mid \lambda x. e$	λ -terms
c	constant (for primitives)
let $x = e$ in e	let-binding
if e then e else e	conditional branching

Primitives

- used for predefined “functions” and “constants”
- **Boolean:** True, False, <, >, =, ...
- **arithmetic:** \times , +, \div , $-$, 0, 1, ...
- **tuples:** Pair, fst, snd
- **lists:** Nil, Cons, head, tail

Types

- **type variables** $\alpha, \alpha_0, \alpha_1, \dots$

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) **function space constructor** \rightarrow

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- **type constructors** C, C_0, C_1, \dots (like List for type of lists)

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- **types** $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- special case – **base types**: Int, Bool (instead of Int(), Bool())

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Example Types

- List(Bool) – list of Booleans
- Pair(Int, Int) – pairs of integers
- Int \rightarrow Int \rightarrow Bool – functions from two integers to Booleans

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Example Types

- List(Bool) – list of Booleans
- Pair(Int, Int) – pairs of integers
- Int \rightarrow Int \rightarrow Bool – functions from two integers to Booleans

A Type for Types

```
data Type = TVar Int
  | TCon String [Type]
deriving Eq
```

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Example Types

- List(Bool) – list of Booleans
- Pair(Int, Int) – pairs of integers
- Int \rightarrow Int \rightarrow Bool – functions from two integers to Booleans

A Type for Types

```
data Type = TVar Int
  | TCon String [Type]
deriving Eq
```

– for easy renaming of **type variables**

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Example Types

- List(Bool) – list of Booleans
- Pair(Int, Int) – pairs of integers
- Int \rightarrow Int \rightarrow Bool – functions from two integers to Booleans

A Type for Types

```
data Type = TVar Int
  | TCon String [Type]
  deriving Eq
```

- for easy renaming of type variables
- **type constructors** and **function type**

Types

- type variables $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) function space constructor \rightarrow
- type constructors C, C_0, C_1, \dots (like List for type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Example Types

- List(Bool) – list of Booleans
- Pair(Int, Int) – pairs of integers
- Int \rightarrow Int \rightarrow Bool – functions from two integers to Booleans

A Type for Types

```
data Type = TVar Int
  | TCon String [Type]
  deriving Eq
```

- for easy renaming of type variables
- type constructors and function type
- **type equality** can be checked

Typing Environments

- set of pairs E , mapping variables and constants to types
- instead of $(e, \tau) \in E$, write $e :: \tau \in E$

Typing Environments

- set of pairs E , mapping variables and constants to types
- instead of $(e, \tau) \in E$, write $e :: \tau \in E$

Typing Judgments

- $E \vdash e :: \tau$
- read: “it can be proved that e is of type τ under E ”

Typing Environments

- set of pairs E , mapping variables and constants to types
- instead of $(e, \tau) \in E$, write $e :: \tau \in E$

Typing Judgments

- $E \vdash e :: \tau$
- read: “it can be proved that e is of type τ under E ”

Examples

- **primitive environment**
 $P = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \text{Nil} :: \text{List}(\alpha), \dots\}$

Typing Environments

- set of pairs E , mapping variables and constants to types
- instead of $(e, \tau) \in E$, write $e :: \tau \in E$

Typing Judgments

- $E \vdash e :: \tau$
- read: “it can be proved that e is of type τ under E ”

Examples

- primitive environment
 $P = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \text{Nil} :: \text{List}(\alpha), \dots\}$
- $P \vdash \text{True} :: \text{Bool}$ – “using primitive environment, it can be shown that True is of type Bool”

Type Substitutions

- **mapping** σ from type variables to types

Type Substitutions

- mapping σ from type variables to types
- **apply substitution** to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- **composition** $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples



domain

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$
- $\tau\sigma_1 =$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$
- $\tau\sigma_1 = \text{List}(\text{List}(\alpha_2))$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$
- $\tau\sigma_1 = \text{List}(\text{List}(\alpha_2))$
- $\sigma_2 = \{\alpha_2 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}\}$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$
- $\tau\sigma_1 = \text{List}(\text{List}(\alpha_2))$
- $\sigma_2 = \{\alpha_2 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}\}$
- $\sigma_1\sigma_2 =$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$
- $\tau\sigma_1 = \text{List}(\text{List}(\alpha_2))$
- $\sigma_2 = \{\alpha_2 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}\}$
- $\sigma_1\sigma_2 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$ (“first apply σ_1 and then σ_2 ”)

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \text{List}(\alpha_2), \alpha_2 \mapsto \text{Bool}\}$
- $\tau = \text{List}(\alpha_1)$
- $\tau\sigma_1 = \text{List}(\text{List}(\alpha_2))$
- $\sigma_2 = \{\alpha_2 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}\}$
- $\sigma_1\sigma_2 = \{\alpha_1 \mapsto \text{List}(\text{Int}), \alpha_2 \mapsto \text{Bool}, \alpha_3 \mapsto \text{Int}\}$

Implementing Type Substitutions

```
type TSub = [(Int, Type)]
```

Implementing Type Substitutions

```
type TSub = [(Int, Type)]
```

“Extracting” Maybes with Default

- `Data.Maybe.fromMaybe :: a -> Maybe a -> a`

Implementing Type Substitutions

```
type TSub = [(Int, Type)]
```

“Extracting” Maybes with Default

- `Data.Maybe.fromMaybe :: a -> Maybe a -> a`
- satisfies equations $\text{fromMaybe } x \text{ Nothing} = x$
 $\text{fromMaybe } x \text{ (Just } y) = y$

Implementing Type Substitutions

```
type TSub = [(Int, Type)]
```

“Extracting” Maybes with Default

- `Data.Maybe.fromMaybe :: a -> Maybe a -> a`
- satisfies equations $\text{fromMaybe } x \text{ Nothing} = x$
 $\text{fromMaybe } x \text{ (Just } y) = y$

Applying Substitutions to Types

```
tsub :: TSub -> Type -> Type
```

```
tsub s (x@(TVar i)) = fromMaybe x $ lookup i s
```

```
tsub s (TCon g ts) = TCon g (map (tsub s) ts)
```


Implementing Type Substitutions

```
type TSub = [(Int, Type)]
```

“Extracting” Maybes with Default

- `Data.Maybe.fromMaybe :: a -> Maybe a -> a`
- satisfies equations `fromMaybe x Nothing = x`
`fromMaybe x (Just y) = y`

Applying Substitutions to Types

```
tsub :: TSub -> Type -> Type
```

```
tsub s (x@(TVar i)) = fromMaybe x $ lookup i s
```

```
tsub s (TCon g ts) = TCon g (map (tsub s) ts)
```

Composition

```
tcomp :: TSub -> TSub -> TSub
```

```
s1 `tcomp` s2 =
```

```
  map (\(x, t) -> (x, tsub s2 t)) s1 ++
```

```
    filter ((`notElem` dom1) . fst) s2
```

```
  where dom1 = map fst s1
```

Type Checking as Natural Deduction Rules

$$\frac{e :: \tau \in E}{e :: \tau\sigma} \text{ (ins)}$$

$$\frac{e_1 :: \tau_2 \rightarrow \tau_1 \quad e_2 :: \tau_2}{e_1 e_2 :: \tau_1} \text{ (app)}$$

$$\frac{\boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e :: \tau_2 \end{array}}}{\lambda x. e :: \tau_1 \rightarrow \tau_2} \text{ (abs)}$$

$$\frac{e_1 :: \tau_1 \quad \boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e_2 :: \tau_2 \end{array}}}{\mathbf{let } x = e_1 \mathbf{ in } e_2 :: \tau_2} \text{ (let)}$$

$$\frac{e :: \tau}{e :: \tau} \text{ (copy)}$$

$$\frac{e_1 :: \mathbf{Bool} \quad e_2 :: \tau \quad e_3 :: \tau}{\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3 :: \tau} \text{ (ite)}$$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$
- 1 $\text{True} :: \text{Bool}$ ins E

$$\frac{e :: \tau \in E}{e :: \tau \sigma} \text{ (ins)}$$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$
- | | | |
|---|------------------------------|-----------------|
| 1 | $\text{True} :: \text{Bool}$ | $\text{ins } E$ |
| 2 | $x :: \text{Bool}$ | assumption |

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall $(\backslash x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$
- 1 $\text{True} :: \text{Bool}$ ins E
- 2 $x :: \text{Bool}$ assumption
- 3 $\lambda x. x :: \text{Bool} \rightarrow \text{Bool}$ abs 2

$$\frac{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e :: \tau_2 \end{array}}{\lambda x. e :: \tau_1 \rightarrow \tau_2} \text{ (abs)}$$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$
- 1 $\text{True} :: \text{Bool}$ ins E
- 2 $x :: \text{Bool}$ assumption
- 3 $\lambda x. x :: \text{Bool} \rightarrow \text{Bool}$ abs 2
- 4 $(\lambda x. x) \text{ True} :: \text{Bool}$ app 3, 1

$$\frac{e_1 :: \tau_2 \rightarrow \tau_1 \quad e_2 :: \tau_2}{e_1 e_2 :: \tau_1} \text{ (app)}$$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$
- | | | |
|---|-------------------------------------------------------|---------------------|
| 1 | $\text{True} :: \text{Bool}$ | $\text{ins } E$ |
| 2 | $x :: \text{Bool}$ | assumption |
| 3 | $\lambda x. x :: \text{Bool} \rightarrow \text{Bool}$ | $\text{abs } 2$ |
| 4 | $(\lambda x. x) \text{ True} :: \text{Bool}$ | $\text{app } 3, 1$ |
- recall
 $\text{double} :: \text{Int} \rightarrow \text{Int}$
 $\text{double } x = x + x$
- prove $E \vdash \lambda x. x + x :: \text{Int} \rightarrow \text{Int}$

Examples

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$

- recall $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$

- prove judgment $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$

- 1 $\text{True} :: \text{Bool}$ ins E
- 2

$x :: \text{Bool}$	assumption
--------------------	------------
- 3 $\lambda x. x :: \text{Bool} \rightarrow \text{Bool}$ abs 2
- 4 $(\lambda x. x) \text{ True} :: \text{Bool}$ app 3, 1

- recall

$\text{double} :: \text{Int} \rightarrow \text{Int}$

$\text{double } x = x + x$

- prove $E \vdash \lambda x. x + x :: \text{Int} \rightarrow \text{Int}$

- 1

$x :: \text{Int}$	assumption
-------------------	------------
- 2 $+ :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ ins E
- 3 $(+) x :: \text{Int} \rightarrow \text{Int}$ app 2, 1
- 4 $x + x :: \text{Int}$ app 3, 1
- 5 $\lambda x. x + x :: \text{Int} \rightarrow \text{Int}$ abs 1-4

Unification and its Implementation

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Problem – Unification

pair of types

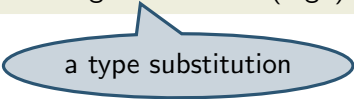
input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE



a type substitution

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

syntactic equality

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is **satisfiable** iff exists σ such that $\tau\sigma = \tau'\sigma$

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- if $\tau \approx \tau'$ satisfiable by σ , then σ is called **solution** of $\tau \approx \tau'$

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- if $\tau \approx \tau'$ satisfiable by σ , then σ is called solution of $\tau \approx \tau'$
- **unification problem** is finite sequence of equations

$$\tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n$$

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- if $\tau \approx \tau'$ satisfiable by σ , then σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n$$

- \square denotes empty sequence

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- if $\tau \approx \tau'$ satisfiable by σ , then σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n$$

- \square denotes empty sequence
- **unification** – solving given unification problem

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier (mgu) σ with $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- if $\tau \approx \tau'$ satisfiable by σ , then σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n$$

- \square denotes empty sequence
- unification – solving given unification problem
- **set of type variables** of a type

$$\mathcal{V}(\alpha) = \{\alpha\}$$

$$\mathcal{V}(\tau_1 \rightarrow \tau_2) = \mathcal{V}(\tau_1) \cup \mathcal{V}(\tau_2)$$

$$\mathcal{V}(C(\tau_1, \dots, \tau_n)) = \mathcal{V}(\tau_1) \cup \dots \cup \mathcal{V}(\tau_n)$$

Unification Rules

(d) decomposition

$$\frac{E_1; C(\tau_1, \dots, \tau_n) \approx C(\tau'_1, \dots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n; E_2} \text{ (d}_1\text{)}$$

$$\frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \text{ (d}_2\text{)}$$

Unification Rules

(d) decomposition

$$\frac{E_1; C(\tau_1, \dots, \tau_n) \approx C(\tau'_1, \dots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n; E_2} \quad (d_1)$$

$$\frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \quad (d_2)$$

(t) removal of **trivial** equations

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2} \quad (t)$$

Unification Rules

(d) decomposition

$$\frac{E_1; C(\tau_1, \dots, \tau_n) \approx C(\tau'_1, \dots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n; E_2} \quad (d_1)$$

$$\frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \quad (d_2)$$

(t) removal of trivial equations

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2} \quad (t)$$

(v) **variable** elimination

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_1)$$

$$\frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_2)$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\text{List}(\text{Bool}) \approx \text{List}(\alpha)$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)} \text{Bool} \approx \alpha$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\text{List}(\text{Bool}) \approx \text{List}(\alpha) \begin{array}{l} \Rightarrow_{\{\}}^{(d_1)} \\ \Rightarrow_{\{v_2\}}^{(v_2)} \end{array} \quad \text{Bool} \approx \alpha$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\begin{array}{l} \text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)} \text{Bool} \approx \alpha \\ \Rightarrow_{\{\alpha \mapsto \text{Bool}\}}^{(v_2)} \end{array}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\begin{array}{l} \text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)} \text{Bool} \approx \alpha \\ \Rightarrow_{\{\alpha \mapsto \text{Bool}\}}^{(v_2)} \square \end{array}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\begin{array}{l} \text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)} \text{Bool} \approx \alpha \\ \Rightarrow_{\{\alpha \mapsto \text{Bool}\}}^{(v_2)} \square \end{array}$$

mgu: $\{\}\{\alpha \mapsto \text{Bool}\}$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_n$$

Example

$$\begin{array}{l} \text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)} \text{Bool} \approx \alpha \\ \Rightarrow_{\{\alpha \mapsto \text{Bool}\}}^{(v_2)} \square \end{array}$$

mgu: $\{\{\alpha \mapsto \text{Bool}\} = \{\alpha \mapsto \text{Bool}\}$

Unification.hs – Implementing Unification

Interface

- input: unification problem `type UP = [(Type, Type)]`

Unification.hs – Implementing Unification

Interface

- input: unification problem `type UP = [(Type, Type)]`
- output: type substitution

Unification.hs – Implementing Unification

Interface

- input: unification problem `type UP = [(Type, Type)]`
- output: type substitution
- unification: `unify :: UP -> TSub`

Unification.hs – Implementing Unification

Interface

- input: unification problem `type UP = [(Type, Type)]`
- output: type substitution
- unification: `unify :: UP -> TSub`

The Free Variables of a Type

```
tvars :: Type -> [Int]
tvars (TVar x)    = [x]
tvars (TCon g ts) = foldr (union . tvars) [] ts
```

Unification.hs – Implementing Unification

Interface

- input: unification problem `type UP = [(Type, Type)]`
- output: type substitution
- unification: `unify :: UP -> TSub`

The Free Variables of a Type

```
tvars :: Type -> [Int]
tvars (TVar x)    = [x]
tvars (TCon g ts) = foldr (union . tvars) [] ts
```

Applying Type Substitutions to Unification Problems

```
tsubUP :: TSub -> UP -> UP
tsubUP s = map (\(l, r) -> (tsub s l, tsub s r))
```

Unification

```
unify :: UP -> TSub
unify eqs = go [] eqs
  where
    go s []          = s
    go s (eq:eqs) =
      go (s `tcomp` s') (eqs' ++ tsubUP s' eqs)
      where
        (eqs', s') = step eq
step :: (Type, Type) -> (UP, TSub)
...
```

Unification

```
unify :: UP -> TSub
unify eqs = go [] eqs
  where
    go s []          = s
    go s (eq:eqs) =
      go (s `tcomp` s') (eqs' ++ tsubUP s' eqs)
      where
        (eqs', s') = step eq
step :: (Type, Type) -> (UP, TSub)
...
```

Removal of Trivial Equations

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2} \text{ (t)}$$

```
step (t, u) | t == u = ([], [])
```

Variable Elimination

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_1) \qquad \frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_2)$$

```
step (TVar x, t) = singletonSub x t
```

```
step (t, TVar x) = singletonSub x t
```

```
where
```

```
notUnif = error "not unifiable"
```

```
singletonSub x t =
```

```
  if x `elem` tvars t then notUnif else ([], [(x, t)])
```

Variable Elimination

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_1) \qquad \frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_2)$$

step (TVar **x**, **t**) = singletonSub **x t**

step (**t**, TVar **x**) = singletonSub **x t**

where

notUnif = error "not unifiable"

singletonSub **x t** =

if **x** `elem` tvars **t** then notUnif else ([], [(**x**, **t**)])

Decomposition

$$\frac{E_1; C(\tau_1, \dots, \tau_n) \approx C(\tau'_1, \dots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n; E_2} \quad (d_1) \qquad \frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \quad (d_2)$$

step (TCon **f ts**, TCon **g us**) |

f == **g** && length **ts** == length **us** = (zip **ts us**, [])

Type Inference

Type Inference Problems

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$ ”

Type Inference Problems

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$ ”

Haskell Example

what is most general type of `let id = \x -> x in id 0`

Type Inference Problems

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$ ”

Haskell Example

what is most general type of `let id = \x -> x in id 0`

Example

- $E = \{0 :: \text{Int}\}$

Type Inference Problems

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$ ”

Haskell Example

what is most general type of `let id = \x -> x in id 0`

Example

- $E = \{0 :: \text{Int}\}$
- $E \triangleright \text{let } id = \lambda x. x \text{ in } id\ 0 :: \alpha_0$

Type Inference Problems

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$ ”

Haskell Example

what is most general type of `let id = \x -> x in id 0`

Example

- $E = \{0 :: \text{Int}\}$
- $E \triangleright \text{let } id = \lambda x. x \text{ in } id\ 0 :: \alpha_0$
- $\sigma = \{\alpha_0 \mapsto \text{Int}\}$

Type Inference Problems

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$ ”

Haskell Example

what is most general type of `let id = \x -> x in id 0`

Example

- $E = \{0 :: \text{Int}\}$
- $E \triangleright \text{let } id = \lambda x. x \text{ in } id\ 0 :: \alpha_0$
- $\sigma = \{\alpha_0 \mapsto \text{Int}\}$

- | | | |
|---|-----------------------------------------------------------------|------------|
| 1 | $x :: \text{Int}$ | assumption |
| 2 | $\lambda x. x :: \text{Int} \rightarrow \text{Int}$ | abs 1 |
| 3 | $id :: \text{Int} \rightarrow \text{Int}$ | assumption |
| 4 | $0 :: \text{Int}$ | ins E |
| 5 | $id\ 0 :: \text{Int}$ | app 3, 4 |
| 6 | $\text{let } id = \lambda x. x \text{ in } id\ 0 :: \text{Int}$ | let 2, 3–5 |

Typing Constraint Rules

$$\frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \text{ (con)}$$

$$\frac{E \triangleright e_1 \ e_2 :: \tau}{E \triangleright e_1 :: \alpha \rightarrow \tau; E \triangleright e_2 :: \alpha} \text{ (app)}$$

$$\frac{E \triangleright \lambda x. e :: \tau}{\tau \approx \alpha_1 \rightarrow \alpha_2; E, x :: \alpha_1 \triangleright e :: \alpha_2} \text{ (abs)}$$

$$\frac{E \triangleright \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 :: \tau}{E \triangleright e_1 :: \alpha; E, x :: \alpha \triangleright e_2 :: \tau} \text{ (let)}$$

$$\frac{E \triangleright \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 :: \tau}{E \triangleright e_1 :: \mathbf{Bool}; E \triangleright e_2 :: \tau; E \triangleright e_3 :: \tau} \text{ (ite)}$$

Typing Constraint Rules

$$\frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \text{ (con)}$$

$$\frac{E \triangleright e_1 e_2 :: \tau}{E \triangleright e_1 :: \alpha \rightarrow \tau; E \triangleright e_2 :: \alpha} \text{ (app)}$$

$$\frac{E \triangleright \lambda x. e :: \tau}{\tau \approx \alpha_1 \rightarrow \alpha_2; E, x :: \alpha_1 \triangleright e :: \alpha_2} \text{ (abs)}$$

$$\frac{E \triangleright \mathbf{let} x = e_1 \mathbf{in} e_2 :: \tau}{E \triangleright e_1 :: \alpha; E, x :: \alpha \triangleright e_2 :: \tau} \text{ (let)}$$

$$\frac{E \triangleright \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3 :: \tau}{E \triangleright e_1 :: \mathbf{Bool}; E \triangleright e_2 :: \tau; E \triangleright e_3 :: \tau} \text{ (ite)}$$

where α in (app) and (let), and α_1 and α_2 in (abs) are **fresh** (do not occur in premises of rule)

Recipe – Type Inference

- to find most general type of e under E

Recipe – Type Inference

- to find most general type of e under E
- first take $E \triangleright e :: \alpha_0$ (for some fresh type variable α_0)

Recipe – Type Inference

- to find most general type of e under E
- first take $E \triangleright e :: \alpha_0$ (for some fresh type variable α_0)
- then, use typing constraint rules to generate unification problem u (if at any point no rule applicable **Not Typable**)

Recipe – Type Inference

- to find most general type of e under E
- first take $E \triangleright e :: \alpha_0$ (for some fresh type variable α_0)
- then, use typing constraint rules to generate unification problem u (if at any point no rule applicable **Not Typable**)
- if u has no solution (none of the unification rules is applicable before reaching \square) then **Not Typable**, otherwise, obtain solution σ

Recipe – Type Inference

- to find most general type of e under E
- first take $E \triangleright e :: \alpha_0$ (for some fresh type variable α_0)
- then, use typing constraint rules to generate unification problem u (if at any point no rule applicable **Not Typable**)
- if u has no solution (none of the unification rules is applicable before reaching \square) then **Not Typable**, otherwise, obtain solution σ
- finally, $\alpha_0\sigma$ is most general type of e

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** $id\ 1$ with respect to P

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright$ **let** $id = \lambda x. x$ **in** id 1 $:: \alpha_0$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright$ **let** $id = \lambda x. x$ **in** id 1 $:: \alpha_0$

$P \triangleright$ **let** $id = \lambda x. x$ **in** id 1 $:: \alpha_0$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright$ **let** $id = \lambda x. x$ **in** id 1 $:: \alpha_0$

$$\underline{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}$$

$$\frac{E \triangleright \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 :: \tau}{E \triangleright e_1 :: \alpha; E, x :: \alpha \triangleright e_2 :: \tau} \text{ (let)}$$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\begin{array}{l} \Rightarrow^{(\text{let})} \frac{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}{P \triangleright \lambda x. x :: \alpha_1;} \\ P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0 \end{array}$$

$$\boxed{\frac{E \triangleright \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 :: \tau}{E \triangleright e_1 :: \alpha; E, x :: \alpha \triangleright e_2 :: \tau} \text{ (let)}}$$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\begin{array}{c} P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0 \\ \hline \Rightarrow^{(\text{let})} \frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0} \end{array}$$

$$\frac{E \triangleright \lambda x. e :: \tau}{\tau \approx \alpha_1 \rightarrow \alpha_2; E, x :: \alpha_1 \triangleright e :: \alpha_2} \text{ (abs)}$$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright$ **let** $id = \lambda x. x$ **in** id 1 $:: \alpha_0$

$$P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$$

\Rightarrow (let)

$$\frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

\Rightarrow (abs)

$$\begin{aligned} \alpha_1 &\approx \alpha_2 \rightarrow \alpha_3; \\ P, x :: \alpha_2 &\triangleright x :: \alpha_3; \\ P, id :: \alpha_1 &\triangleright id \ 1 :: \alpha_0 \end{aligned}$$

$\frac{E \triangleright \lambda x. e :: \tau}{\tau \approx \alpha_1 \rightarrow \alpha_2; E, x :: \alpha_1 \triangleright e :: \alpha_2} \text{ (abs)}$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\frac{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}{}$$

$$\Rightarrow_{(\text{let})} \frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;}{P, x :: \alpha_2 \triangleright x :: \alpha_3;}$$
$$\frac{P, x :: \alpha_2 \triangleright x :: \alpha_3;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$\frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \text{ (con)}$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\underline{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{let})} \frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;}{P, x :: \alpha_2 \triangleright x :: \alpha_3;}$$
$$\frac{P, x :: \alpha_2 \triangleright x :: \alpha_3;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{con})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$\frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \text{ (con)}$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\underline{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{let})} \frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;}{P, x :: \alpha_2 \triangleright x :: \alpha_3;}$$
$$\frac{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{con})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\frac{E \triangleright e_1 \ e_2 :: \tau}{E \triangleright e_1 :: \alpha \rightarrow \tau; E \triangleright e_2 :: \alpha} \text{ (app)}$$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** $id\ 1$ with respect to P
- start from $P \triangleright \mathbf{let}\ id = \lambda x. x\ \mathbf{in}\ id\ 1 :: \alpha_0$

$$\frac{}{P \triangleright \mathbf{let}\ id = \lambda x. x\ \mathbf{in}\ id\ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{let})} \frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;}{P, x :: \alpha_2 \triangleright x :: \alpha_3;}$$
$$\frac{}{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{con})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{app})} \frac{}{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}$$
$$P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0;$$
$$P, id :: \alpha_1 \triangleright 1 :: \alpha_4$$

$$\frac{E \triangleright e_1\ e_2 :: \tau}{E \triangleright e_1 :: \alpha \rightarrow \tau; E \triangleright e_2 :: \alpha} \text{ (app)}$$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\underline{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{let})} \frac{P \triangleright \lambda x. x :: \alpha_1;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;}{P, x :: \alpha_2 \triangleright x :: \alpha_3;}$$
$$\frac{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{con})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}$$

$$\Rightarrow_{(\text{app})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0;}$$
$$\underline{P, id :: \alpha_1 \triangleright 1 :: \alpha_4}$$

$\frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \text{ (con)}$

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\begin{aligned}
 & \frac{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0}{\Rightarrow (\mathbf{let})} \\
 & \frac{P \triangleright \lambda x. x :: \alpha_1; \quad P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{\Rightarrow (\mathbf{abs})} \\
 & \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \quad P, x :: \alpha_2 \triangleright x :: \alpha_3; \quad P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{\Rightarrow (\mathbf{con})} \\
 & \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3; \quad P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{\Rightarrow (\mathbf{app})} \\
 & \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3; \quad P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0; \quad P, id :: \alpha_1 \triangleright 1 :: \alpha_4}{\Rightarrow (\mathbf{con})^2} \\
 & \alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \mathbf{Int} \approx \alpha_4
 \end{aligned}$$

$ \frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \quad (\mathbf{con}) $

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** id 1 with respect to P
- start from $P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0$

$$\begin{array}{l}
 \Rightarrow^{(\text{let})} \frac{P \triangleright \mathbf{let} \ id = \lambda x. x \ \mathbf{in} \ id \ 1 :: \alpha_0 \quad \alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \underline{\alpha_2 \approx \alpha_3}; \quad \alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \underline{\text{Int} \approx \alpha_4}}{P \triangleright \lambda x. x :: \alpha_1; \quad P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0} \\
 \Rightarrow^{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \quad P, x :: \alpha_2 \triangleright x :: \alpha_3; \quad P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{\alpha_1 \approx \alpha_3 \rightarrow \alpha_3; \alpha_1 \approx \text{Int} \rightarrow \alpha_0} \\
 \Rightarrow^{(\text{con})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3; \quad P, id :: \alpha_1 \triangleright id \ 1 :: \alpha_0}{\alpha_3 \rightarrow \alpha_3 \approx \text{Int} \rightarrow \alpha_0} \\
 \Rightarrow^{(\text{app})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3; \quad P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0; \quad P, id :: \alpha_1 \triangleright 1 :: \alpha_4}{\alpha_3 \approx \text{Int}; \underline{\alpha_3 \approx \alpha_0}} \\
 \Rightarrow^{(\text{con})^2} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3; \quad \alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \text{Int} \approx \alpha_4}{\alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \text{Int} \approx \alpha_4}
 \end{array}$$

$$\Rightarrow^{(v)^2} \{\alpha_2 \mapsto \alpha_3, \alpha_4 \mapsto \text{Int}\}$$

$$\Rightarrow^{(v)} \{\alpha_1 \mapsto \alpha_3 \rightarrow \alpha_3\}$$

$$\Rightarrow^{(d)}$$

$$\Rightarrow^{(v)^2} \{\alpha_0 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}\}$$

□

Exercise

- find most general type of **let** $id = \lambda x. x$ **in** $id\ 1$ with respect to P
- start from $P \triangleright \mathbf{let}\ id = \lambda x. x\ \mathbf{in}\ id\ 1 :: \alpha_0$

$$\begin{array}{l}
 \Rightarrow^{(\text{let})} \frac{P \triangleright \mathbf{let}\ id = \lambda x. x\ \mathbf{in}\ id\ 1 :: \alpha_0}{P \triangleright \lambda x. x :: \alpha_1;} \quad \alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \underline{\alpha_2 \approx \alpha_3}; \\
 \alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \underline{\text{Int} \approx \alpha_4} \\
 \Rightarrow^{(\text{abs})} \frac{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;} \quad \Rightarrow^{(v)^2} \{\alpha_2 \mapsto \alpha_3, \alpha_4 \mapsto \text{Int}\} \\
 \alpha_1 \approx \alpha_3 \rightarrow \alpha_3; \alpha_1 \approx \text{Int} \rightarrow \alpha_0 \\
 \Rightarrow^{(\text{con})} \frac{P, x :: \alpha_2 \triangleright x :: \alpha_3;}{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0} \quad \Rightarrow^{(v)} \{\alpha_1 \mapsto \alpha_3 \rightarrow \alpha_3\} \\
 \alpha_3 \rightarrow \alpha_3 \approx \text{Int} \rightarrow \alpha_0 \\
 \Rightarrow^{(\text{app})} \frac{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;} \quad \Rightarrow^{(d)} \\
 \alpha_3 \approx \text{Int}; \underline{\alpha_3 \approx \alpha_0} \\
 \Rightarrow^{(\text{con})^2} \frac{P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0;}{P, id :: \alpha_1 \triangleright 1 :: \alpha_4} \quad \Rightarrow^{(v)^2} \{\alpha_0 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}\} \\
 \square \\
 \Rightarrow^{(\text{con})^2} \frac{P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0;}{P, id :: \alpha_1 \triangleright 1 :: \alpha_4} \quad \text{mgu: } \{\alpha_0 \mapsto \text{Int}, \alpha_1 \mapsto \text{Int} \rightarrow \text{Int}, \\
 \alpha_2 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}, \alpha_4 \mapsto \text{Int}\}
 \end{array}$$

Exams

1st Exam

- 12:15–14:00 in HSB 1 on February 1, 2019
- online [registration](#) required before 23:59 on January 18, 2019

Exams

1st Exam

- 12:15–14:00 in HSB 1 on February 1, 2019
- online [registration](#) required before 23:59 on January 18, 2019

2nd Exam

- 12:15–14:00 in HSB 1 on March 1, 2019
- online [registration](#) required before 23:59 on February 15, 2019

Exams

1st Exam

- 12:15–14:00 in HSB 1 on February 1, 2019
- online [registration](#) required before 23:59 on January 18, 2019

2nd Exam

- 12:15–14:00 in HSB 1 on March 1, 2019
- online [registration](#) required before 23:59 on February 15, 2019

3rd Exam

- 12:15–14:00 in HSB 1 on September 27, 2019
- online [registration](#) required before 23:59 on September 13, 2019

Exams

1st Exam

- 12:15–14:00 in HSB 1 on February 1, 2019
- online [registration](#) required before 23:59 on January 18, 2019

2nd Exam

- 12:15–14:00 in HSB 1 on March 1, 2019
- online [registration](#) required before 23:59 on February 15, 2019

3rd Exam

- 12:15–14:00 in HSB 1 on September 27, 2019
- online [registration](#) required before 23:59 on September 13, 2019

Some Old Exams

- exam of February 2, 2018 + solutions
- exam of March 2, 2018 + solutions

Homework (for January 25th)

1. Read the [lecture notes about type checking and type inference](#).
2. Use type checking to prove $\{f :: \alpha \rightarrow \beta, y :: \alpha\} \vdash (\lambda x. f x) y :: \beta$.
3. Solve the unification problem
 $\alpha_2 \approx \alpha_1 \rightarrow \alpha_0; \alpha_3 \approx \text{Int} \rightarrow \alpha_2; \alpha_2 \approx \alpha_3$
and compute the resulting mgu, if possible. Check your result using [unify](#).
4. Solve the unification problem $\text{Pair}(\text{Bool}, \alpha_0) \approx \text{Pair}(\alpha_1, \text{Int})$. Check your solution using [unify](#).
5. Use type inference to obtain the most general type of the expression $\text{map } (\lambda x. x)$ with respect to the environment
 $E = \{\text{map} :: (\alpha \rightarrow \beta) \rightarrow \text{List}(\alpha) \rightarrow \text{List}(\beta)\}$.
6. Given the following types for environments and inference problems
`type Env = [(String, Type)]`
`type IP = [(Env, Exp, Type)]`
implement a function `toUp :: IP -> UP` that transforms a given type inference problem into the corresponding unification problem according to our typing constraint rules.