Types

Types are good

Detect error at compile time
Types

Types are good
Detect error at compile time

Types are bad
Tedious to write down
Let’s start from

simPL
Type variables
Our types...
Our types...
Our types...
Type variables

Our types...

int

bool

T → T
Type variables

Our types...

\[
\text{int} \\
\text{bool} \\
T \rightarrow T \\
X
\]
Type variables

Our types...

int
bool
T → T
X

Type variable
fun \{A \to A\} \ x \to x : A \to A
Type variables

Everything good but...
Everything good but...

Writing down types is a pain...
Let's have lists!

Type variables
Let's have lists!

Type variables

Type inference
What if...

Type

inference
What if...

Type inference

let {int} f = fun {int→int} x → x end in {bool} (f 4) < 2 end
What if...

let {int} f = fun {int→int} x → x end in {bool}
(f 4) < 2
end

Could simply be...
What if...

let \{\text{int}\} f = \text{fun} \{\text{int} \rightarrow \text{int}\} x \rightarrow x \text{ end in } \{\text{bool}\}

(f 4) < 2
end

Could simply be...

let f = \text{fun} x \rightarrow x \text{ end in }

(f 4) < 2
end
What if...

let \( \{ \text{int} \} f = \text{fun} \ \{ \text{int} \rightarrow \text{int} \} \ x \rightarrow x \ \text{end} \ \text{in} \ \{ \text{bool} \} \)

\((f \ 4) < 2\)
end

Could simply be...

let \( f = \text{fun} \ x \rightarrow x \ \text{end} \ \text{in} \)

\((f \ 4) < 2\)
end

...and still be type checked?
Indeed it is possible!
Indeed it is possible!

Hindley-Milner type inference
How does it work?
First step, attribute different type variables to the expressions.
let f = fun x → x end in (f 4) < 2 end
let f = fun x → x end in (f 4) < 2 end
let f = fun x → x end in (f 4) < 2 end

let {A} f = fun {B→C} x → x end in {D} (f 4) < 2 end
Second step, generate constraints from the expressions.
let \{A\} f = \text{fun} \{B \rightarrow C\} \ x \rightarrow \ x \ \text{end} \ \text{in} \ \{D\} \ (f \ 4) < 2 \ \text{end}
let \( \{A\} \ f = \text{fun} \ \{B \to C\} \ x \to x \ \text{end in} \ \{D\} \ (f \ 4) < 2 \ \text{end} \)
A = B \rightarrow C

let \{A\} f = fun \{B \rightarrow C\} x \rightarrow x end in \{D\}
(f 4) < 2
end

\Gamma : \text{Id} \rightarrow \text{Type}
Type inference

\[ A = B \to C \]

\[ \Gamma(x) = B \]

\[ \Gamma : \text{Id} \to \text{Type} \]

let \( \{A\} \) f = fun \( \{B \to C\} \) x \( \to \) x end in \( \{D\} \)

(f 4) < 2

end
A = B → C

let {A} f = fun {B→C} x → x end in {D}
(f 4) < 2
end

B = int

Γ(f) = B → C

Γ(x) = B

Γ : Id → Type

B = C

Γ : Id → Type
A = B → C

let {A} f = fun {B→C} x → x end in {D}

(f 4) < 2

end

B = int

C = int

Γ : Id → Type

Γ(x) = B

B = C

Γ(f) = B → C

C = int
\[ \Gamma : \text{Id} \to \text{Type} \]

\[ \Gamma(x) = B \]

\[ \Gamma(f) = B \to C \]

\[ \text{B} = \text{int} \]

\[ \text{C} = \text{int} \]

\[ \text{D} = \text{bool} \]

\[ \text{A} = \text{B} \to \text{C} \]

\[ \text{let } \{\text{A}\} f = \text{fun } \{\text{B} \to \text{C}\} x \to x \text{ end in } \{\text{D}\} \]

\[ (f \ 4) < 2 \]
Third step, solve the constraints.

Using the unification algorithm.
A = B → C, B = C, B = int, C = int, D = bool
Type inference

A = B → C, B = C, B = int, C = int, D = bool
A = B → C, B = C, B = int, C = int, D = bool

A = int → int, B = int, C = int, D = bool
Last step, substitute type variables in the expression.
Type inference

A = int → int, B = int, C = int, D = bool
Type inference

A = int → int, B = int, C = int, D = bool
Type inference

\[ A = \text{int} \rightarrow \text{int}, \ B = \text{int}, \ C = \text{int}, \ D = \text{bool} \]

\[
\begin{align*}
\text{let } & \{\text{int}\rightarrow\text{int}\} \ f = \text{fun} \ \{\text{int}\rightarrow\text{int}\} \ x \ \rightarrow \ x \ \text{end in } \{\text{bool}\} \\
& (f \ 4) < 2 \\
\text{end}
\end{align*}
\]
Type inference

\[ A = \text{int} \rightarrow \text{int}, B = \text{int}, C = \text{int}, D = \text{bool} \]

\[
\text{let } \{\text{int} \rightarrow \text{int}\} \ f = \text{fun} \ \{\text{int} \rightarrow \text{int}\} \ x \rightarrow x \ \text{end in} \ \{\text{bool}\}
\]

\[(f \ 4) < 2\]

end

\[\text{NOT BAD}\]
What about this one?

Type inference
Type inference

What about this one?

let f = fun x → x + 1 end in
(f true)
end
let f = fun x \rightarrow x + 1 end in
(f true)
end

What about this one?

Type inference
let \( f = \text{fun } x \rightarrow x + 1 \text{ end in} \)
\((f \text{ true})\)
end

let \{A\} f = \text{fun } \{B \rightarrow C\} x \rightarrow x + 1 \text{ end in } \{D\} \)
\((f \text{ true})\)
end
What about this one?

let \{A\} f = fun \{B \rightarrow C\} x \rightarrow x + 1 \text{ end in } \{D\} 
(f \text{ true})
end
Type inference

What about this one?

\[ A = B \rightarrow C \]

let \( \{ A \} \) \( f = \) fun \( \{ B \rightarrow C \} \) \( x \rightarrow x + 1 \) end in \( \{ D \} \)

(f true)
end
What about this one?

\[
\begin{align*}
A &= B \to C \\
C &= \text{int}
\end{align*}
\]

\[
\text{let } \{A\} f = \text{fun } \{B \to C\} x \to x + 1 \text{ end in } \{D\}
\]

(f true)
end
let \{A\} f = fun \{B \to C\} x \to x + 1 end in \{D\}

(f true)

end

What about this one?

A = B \to C
C = \text{int}
B = \text{int}

Type inference
Type inference

What about this one?

\[
\begin{align*}
A &= B \rightarrow C \\
C &= \text{int} \\
B &= \text{int}
\end{align*}
\]

let \{A\} f = fun \{B \rightarrow C\} x \rightarrow x + 1 end in \{D\} (f true) end

B = \text{bool}
What about this one?

\[
\begin{align*}
A &= B \rightarrow C \\
C &= \text{int} \\
B &= \text{int}
\end{align*}
\]

let \{A\} f = fun \{B \rightarrow C\} x \rightarrow x + 1 end in \{D\}

(f true) \quad D = B

end

B = \text{bool}
Type inference

What about this one?

\[ A = B \rightarrow C, \ B = \text{int}, \ C = \text{int}, \ B = \text{bool}, \ D = C \]
What about this one?

\[ A = B \rightarrow C, \ B = \text{int}, \ C = \text{int}, \ B = \text{bool}, \ D = C \]
What about this one?

\[ A = B \rightarrow C, \ B = \text{int}, \ C = \text{int}, \ B = \text{bool}, \ D = C \]

Impossible to solve!
Type inference

What about this one?

\[
A = B \rightarrow C, \ B = \text{int}, \ C = \text{int}, \ B = \text{bool}, \ D = C
\]

Impossible to solve!

We reject the program, as it is not well typed.
Type inference

And this one?
Type inference

And this one?

let id = fun x → x end in
(id true) & (id 1) < 2
end
let id = fun x -> x end in
(id true) & (id 1) < 2
end
let id = fun x -> x end in (id true) & (id 1) < 2 end

let \{A\} id = fun \{B \rightarrow C\} x -> x end in \{D\} (id true) & (id 1) < 2 end
Type inference

And this one?

```
let {A} id = fun {B \to C} x \to x end in {D} (id true) & (id 1) < 2 end
```
And this one?

\[
A = B \rightarrow C
\]

\[
\text{let } \{A\} \text{id} = \text{fun } \{B \rightarrow C\} \ x \rightarrow x \text{ end in } \{D\} \ (\text{id true}) \& (\text{id 1}) < 2 \text{ end}
\]
And this one?

A = B → C

B = C

let {A} id = fun {B → C} x → x end in {D} (id true) & (id 1) < 2
end
And this one?

\[
\begin{align*}
A &= B \rightarrow C \\
\text{let } \{A\} \text{ id } &= \text{fun } \{B \rightarrow C\} \ x \rightarrow x \text{ end in } \{D\} \\
(id \text{ true}) \ &\& \&\ (id \text{ 1}) < 2 \\
\text{end} \\
B &= \text{bool}
\end{align*}
\]
Type inference

And this one?

\[
\text{let } \{A\} \text{ id } = \text{fun } \{B \to C\} x \to x \text{ end in } \{D\} (\text{id } \text{true}) \& (\text{id } 1) < 2 \text{ end}
\]

\[
A = B \to C \quad \quad B = C
\]

\[
B = \text{bool} \quad \quad D = \text{bool}
\]
Type inference

And this one?

\[
\begin{align*}
A &= B \to C \\
B &= C \\
B &= \text{bool} \\
B &= \text{int} \\
D &= \text{bool} \\
\text{id} &= \text{fun} \ (B \to C) \ x \to x \ \text{end} \ \text{in} \ {D} \\
(\text{id} \ \text{true}) &\land (\text{id} \ 1) < 2 \\
\end{align*}
\]
And this one?

\[ A = B \rightarrow C, \ B = C, \ B = \text{int}, \ B = \text{bool}, \ D = \text{bool} \]
Type inference

And this one?

$$A = B \rightarrow C, \quad B = C, \quad B = \text{int}, \quad B = \text{bool}, \quad D = \text{bool}$$
And this one?

\[ A = B \rightarrow C, \ B = C, \ B = \text{int}, \ B = \text{bool}, \ D = \text{bool} \]

Impossible to solve!
Type inference

And this one?

\[ A = B \rightarrow C, \ B = C, \ B = \text{int}, \ B = \text{bool}, \ D = \text{bool} \]

Impossible to solve!
Type inference

But, this program would make sense...
But, this program would make sense...

let id = fun x → x end in
(id true) & (id 1) < 2
end
Let's just do like this, right?
Type inference

Let’s just do like this, right?

let
    idInt = fun x → x end
    idBool = fun x → x end
in
    (idBool true) & (idInt 1) < 2
end
Let's just do like this, right?

let

```
let idInt = fun x -> x end
let idBool = fun x -> x end

in

(idBool true) & (idInt 1) < 2
```

end
Abstraction principle
Abstraction principle

“Each significant piece of functionality in a program should be implemented in just one place in the source code.”
Abstraction principle

“Each significant piece of functionality in a program should be implemented in just one place in the source code.

Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.”
Let's have lists!

Type variables

Type inference
<table>
<thead>
<tr>
<th>Type variables</th>
<th>Type inference</th>
<th>Let polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We want to treat let expressions differently

The instances of those declarations should be

independently typed
Let polymorphism

Type Scheme
Polymorphism

Type Scheme

\[ \forall X_1..X_n . T \]
Let polymorphism

Type Scheme

\[ \forall X_1 \ldots X_n. T \]
Let polymorphism

Type Scheme

∀ X₁..Xₙ . T

Type

Type Variables
\[ \Gamma : \text{Id} \rightarrow \text{Type Scheme} \]

Type Scheme

\[ \forall X_1..X_n . T \]
Let
polymorphism

Type Scheme
Let polymorphism

Type Scheme

∀ A . A → A
Let

polymorphism

Type Scheme

∀ A . A → A  ∀ A, B . B → A
Let polymorphism

Type Scheme

\( \forall A . A \rightarrow A \quad \forall A, B . B \rightarrow A \)

\( \forall A . B \rightarrow A \rightarrow \text{int} \)
Let polymorphism

Generalization
Generalization

Let polymorphism

A → A
Generalization

Let polymorphism

$$A \rightarrow A$$
Let

polymorphism

Generalization

\[ A \rightarrow A \]

\[ \forall A . A \rightarrow A \]
Generalization

Let polymorphism

\( A \rightarrow A \)

\( \forall A . \ A \rightarrow A \)

Type

Type Scheme
Generalization

Do not generalize type variables that are found in the type environment!
Let

polymorphism

Instantiation
Let polymorphism

∀ A . A → A

Instantiation
Instantiation

∀ A . A → A
Let polymorphism

Instantiation

$\forall A . A \rightarrow A$

Type Scheme

$B \rightarrow B$
Let polymorphism

Instantiation

∀ A . A → A

B → B

Type Scheme

Type
Instantiation

\[ \forall A . A \rightarrow A \]

Type Scheme

\[ C \rightarrow C \]

Type

Let

polymorphism
Let Polymorphism
First step
Finding the principal type of the right hand side
First step
Finding the principal type of the right hand side

let id = fun x → x end in ... end
First step
Finding the principal type of the right hand side

let id = fun x → x end in ... end

A → A
Let
polymorphism

Second step
Generalizing the type
Let polymorphism

**Second step**
Generalizing the type

A $\rightarrow$ A
Let polymorphism

Second step
Generalizing the type

\[ A \to A \quad \forall A \cdot A \to A \]
Let polymorphism

Third step
Extending the type environment
Let polymorphism

**Third step**
Extending the type environment

\[ \Gamma \ (\text{id}) = \forall A . \ A \rightarrow A \]
When seeing an identifier

We instantiate its scheme to get its type
When seeing an identifier
We instantiate its scheme to get its type

... in

(id 2) < (id 3) & (id true)
end
... in
  (id 2) < (id 3) & (id true)
end

polymorphism
Let polymorphism

\[ \Gamma \text{ (id)} = \forall A . A \rightarrow A \]

... in

\((\text{id 2}) < (\text{id 3}) \land (\text{id true})\)

end
\( \Gamma (\text{id}) = \forall A . A \rightarrow A \)

Let polymorphism

\[ B \rightarrow B \]

... in

\( (\text{id}\ 2) < (\text{id}\ 3) \& (\text{id}\ \text{true}) \)

end
Let polymorphism

\[ \Gamma (id) = \forall A . A \rightarrow A \]

\[ \begin{align*}
B & \rightarrow B \\
C & \rightarrow C \\
\text{... in} & \\
(id 2) & < (id 3) \& (id \text{ true}) \\
\text{end}
\end{align*} \]
Let polymorphism

\[ \Gamma \text{(id)} = \forall A . A \rightarrow A \]

\[
\begin{align*}
B & \rightarrow B \\
C & \rightarrow C \\
D & \rightarrow D \\
\end{align*}
\]

... in

(id 2) < (id 3) & (id true)

end
Type 
variables

Type
inference

Let
polymorphism
let twice = fun f x → (f (f x)) end

in

(twice fun x → x + 1 end 2) = 4 &
(twice fun x → \x end true)

end
tyPL still lacks something fundamental compared to rePL

Data Structures
polymorphism

Type variables

Type inference

liPL
Type variables

Type inference

Let polymorphism
Type variables

Type inference

Let polymorphism

Let's have lists!
Let’s have lists!
Let's have lists!
Let's have lists!

- Empty List: `[]`
- List constructor: `1 :: 2 :: 3 :: []`
Let’s have lists!

- Empty List: `[]`
- List constructor: `[1 :: 2 :: 3 :: []]`
- Syntactic sugar: `[1, 2, 3]`
Let’s have lists!
Let’s have lists!

(h:ead [1, 2, 3])
Let's have lists!

(head [1, 2, 3])
(tail [1, 2, 3])

[2, 3]
Let's have lists!

(head [1, 2, 3]) (tail [1, 2, 3]) (empty [2, 3])

1 [2, 3] false
Lists are homogeneous

Let's have lists!
Lists are homogeneous.

All elements have the same type.
Lists are **homogeneous**

All elements have the same type.

\[ [1, 2, 3] : [\text{int}] \]
Lists are homogeneous

All elements have the same type.

\[[1, 2, 3] : [\text{int}]\] \hspace{2cm} \[[\text{true}] : [\text{bool}]\]

Let’s have lists!
Lists are **homogeneous**

All elements have the same type.

\[ [1, 2, 3] : [\text{int}] \]
\[ [\text{true}] : [\text{bool}] \]

\[ [] : [\text{A}] \]
Let's have lists!
Type variables
Type inference
Let polymorphism
Let’s have lists!
let map = recfun map f xs →
  if (empty xs) then
    []
  else
    (f (head xs)) :: (map f (tail xs))
  end
end

in
  (map fun x → x + 2 end [1, 2, 3])
end
Let's have lists!

Type variables
Type inference
Let polymorphism

Who has the first question?
Let's have lists!

Type variables
Type inference
Let polymorphism

Thank you for your attention!