A type-system for Nix

Théophane Hufschmitt

October 28 2017

Motivation

At the very begining…

Nix won't be complete until it has static typing[1]

Maintenance needs

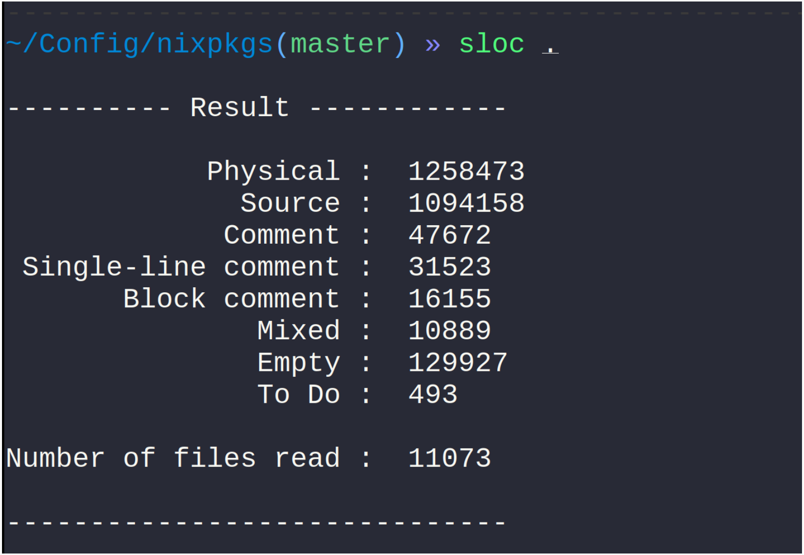
- nixpkgs: 1M sloc
- Errors hard to spot

[1] Eelco Dolstra

Why isn't it done yet

```
lst:
  let
    x = head lst;
    y = elemAt lst 1;
  in
  if isString x
  then y.${x}
  else x + y
```

Impossible to type everything



```
~/Config/nixpkgs(master) » sloc .

---------- Result ------------

            Physical :  1258473
              Source :  1094158
             Comment :  47672
 Single-line comment :  31523
       Block comment :  16155
               Mixed :  10889
               Empty :  129927
               To Do :  493

Number of files read :  11073

------------------------------
```

Requirements

- No compilation
- No syntax extension
- Type as much code as possible
- The ill-typed code must still be accepted

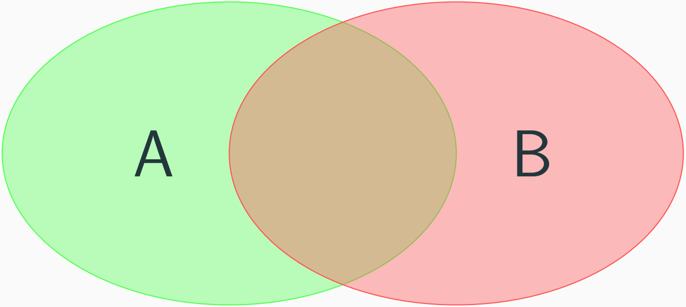
Need for powerful types

```
let
  f = x: y: if isInt x then x + y else x && y;
in f
```

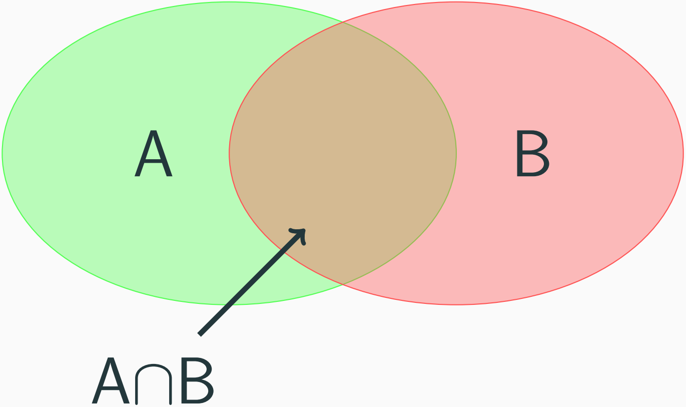
→ Type of f?

Int → **Int** → **Int**, but also **Bool** → **Bool** → **Bool**

Set-theory to the rescue



Set-theory to the rescue



We can do the same with types

(more or less)

- $\cup, \cap, \setminus, \subseteq \rightarrow \vee, \wedge, \setminus, \leq$
- Singleton types 1, **true**, "blah", ...

Back to our example

```
let
  f = x: y: if isInt x then x + y else x && y;
in f
```

f is of type (**Int** $\rightarrow$ **Int** $\rightarrow$ **Int**) $\wedge$ (**Bool** $\rightarrow$ **Bool** $\rightarrow$ **Bool**)

Gradual type

Let's introduce "?"

- Represents unknown types
- Used to type untypeable expressions

```
let x = getEnv "X"; in {y = 1}.${x}
```

Inference alone not always enough

```
x : x+1
» ? → Int

x :
  if isInt x then −x else not x
» ? → (Int ∨ Bool)
```

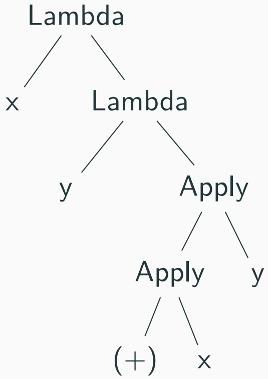
Inference alone not always enough

```
x /*: Int */ : x+1
» Int → Int

x :
  if isInt x then −x else not x
» ? → (Int ∨ Bool)
```

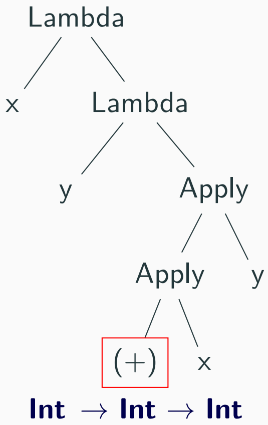
Inference alone not always enough

```
x /*: Int */ : x+1
» Int → Int

x /*: Int ∨ Bool*/ :
  if isInt x then −x else not x
» Int ∨ Bool → (Int ∨ Bool)
```

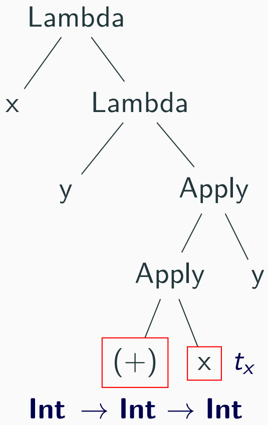
Type reconstruction



Type reconstruction

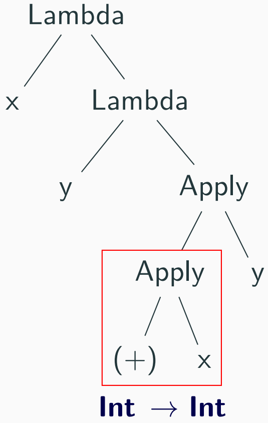


Type reconstruction



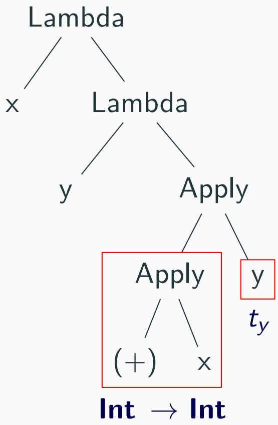
Type reconstruction

$t_x = \textbf{Int}$



Type reconstruction

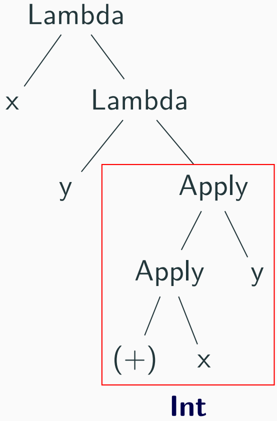
$t_x = \mathbf{Int}$



Type reconstruction

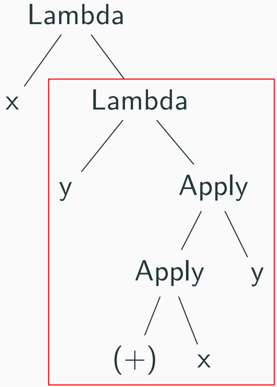
$t_x = \textbf{Int}$

$t_y = \textbf{Int}$



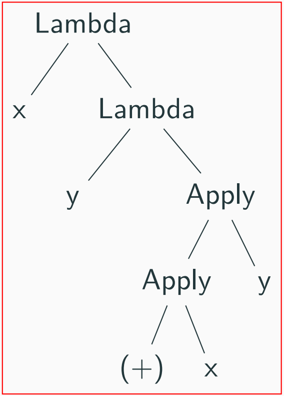
Type reconstruction

$t_x = \textbf{Int}$



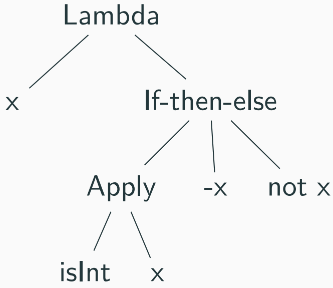
$\textbf{Int} \rightarrow \textbf{Int}$

Type reconstruction

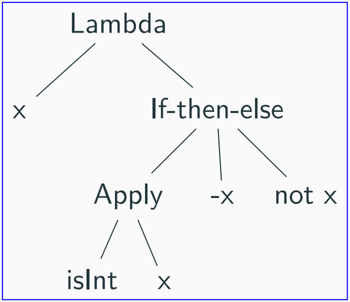


Int → **Int** → **Int**

Type checking

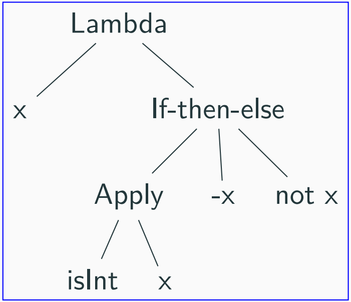


Type checking



$(\mathbf{Int} \rightarrow \mathbf{Int}) \land (\mathbf{Bool} \rightarrow \mathbf{Bool})$

Type checking



Lambda

x

If-then-else

Apply

-x

not x

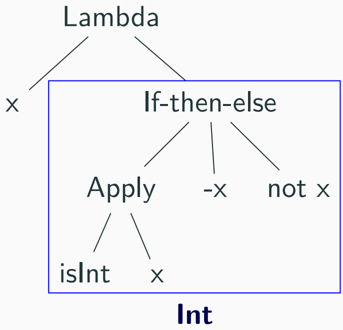
isInt

x

Int $\to$ **Int**

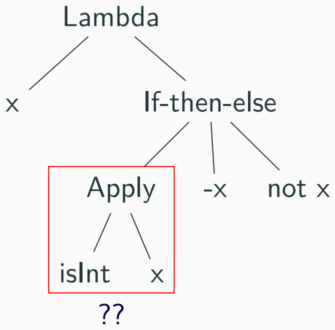
Type checking

$t_x = \mathbf{Int}$



Type checking

$t_x = \mathbf{Int}$


Lambda

x

If-then-else

Apply

-x

not x

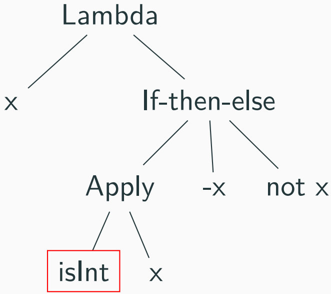
isInt

x

??

Type checking

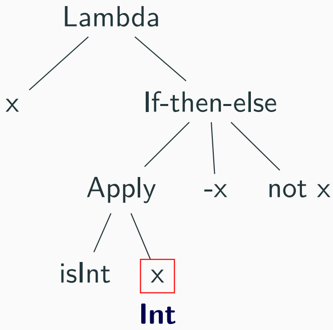
$t_x = \textbf{Int}$



$$(\textbf{Int} \to \textbf{true}) \land (\neg\textbf{Int} \to \textbf{false})$$

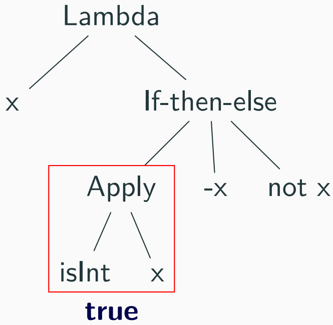
Type checking

$t_x = \textbf{Int}$



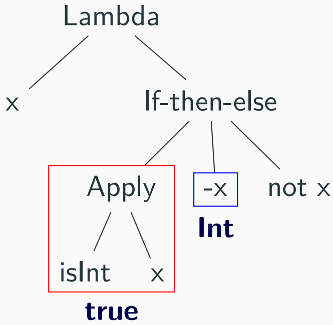
Type checking

$t_x = \textbf{Int}$



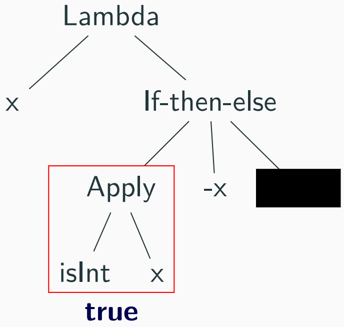
Type checking

$t_x = \mathbf{Int}$

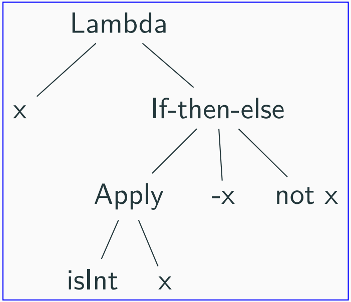


Type checking

$t_x = \textbf{Int}$

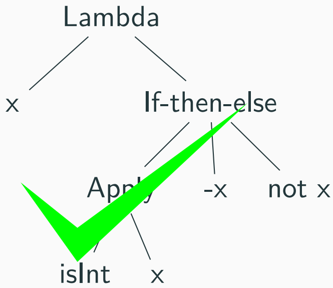


Type checking



Bool → **Bool**

Type checking



Checking to the rescue

```
let f /*: (Int → Int) ∧ (Bool → Bool) */
  = x: if isInt x then −x else not x;
in f
» (Int → Int) ∧ (Bool → Bool)
```

More precision

```
let f = x /*: Int */ : ((y: y) x /*: Bool*/)); in f
```

→ Pass

More precision

```
let f /*: Int → Bool */ = x : ((y: y) x )); in f
```

→ Error

Summary of the features

- Types in comments in normal nix code
- (Hopefully) powerful enough type-system
- Lax by default and safe when needed

  `x: e` ⇔ `x /*: ? */: e`

Lists

Regular expression lists

```
[ 1 2 true ] /*: [ Int* true "bar"# ] */
[ true "bar" ] /*: [ Int* true "bar"# ] */
```

Lists

Regular expression lists

```
[ 1 2 true ] /*: [ Int* true "bar"# ] */
[ true "bar" ] /*: [ Int* true "bar"# ] */

[ Int Bool ] ≈ (Int, Bool)
```

Static attribute sets

```
{ x = 1; y = false; z = "foo" }
```

Static attribute sets

```
{ x = 1; y = false; z = "foo" }
```

```
» { x = 1; y = false; z = "foo" }
```

More attribute sets

```
{ x /*: Int */
, y /*: Int */ ? 1
, ... }:
  x + y
```

More attribute sets

```
{ x /*: Int */
, y /*: Int */ ? 1
, ... }:
  x + y
```

```
» { x = Int , y =? Int , .. } → Int
```

Dynamic attribute sets

```
let
  myFunction /*: Int → String */ = …;
  x = getEnv "Foo";
in
{ ${x} = 1; ${myFunction 2} = true }
```

Dynamic attribute sets

```
let
  myFunction /*: Int → String */ = …;
  x = getEnv ”Foo”;
in
{ ${x} = 1; ${myFunction 2} = true }

» { _ =? 1 ∨ true }
```

Gradual type sometimes unwanted

`(x: x)` is basically an unsafe cast

→ We would sometimes like to have more guaranties

- Don't automatically add gradual types everywhere
- Or even disable the gradual type

Strict mode

Gradual type

```
((x: x) 1)  /*: Bool */
```

Typechecks

Strict mode

Gradual type

```
((x: x) 1) /*# strict-mode */ /*: Bool */
```

Error

Strict mode

Gradual type

```
((x: x) 1) /*# strict-mode */ /*: Bool */
```

Error

Records definition

```
let
  x = getEnv ”FOO”;
  y = getEnv ”BAR”;
in
{ ${x} = 1; ${y} = 2; }
```

Typechecks

Strict mode

Gradual type

```
((x: x) 1) /*# strict-mode */ /*: Bool */
```

Error

Records definition

```
let
  x = getEnv "FOO";
  y = getEnv "BAR";
in
{ ${x} = 1; ${y} = 2; } /*# strict-mode */
```

Error

Control of the gradual type

```
let
  cast = x: x;
in
(cast 1)
    /*: Bool */
```

Typechecks

Control of the gradual type

```
let
  cast = x: x;
in
(cast 1)
  /*# no-gradual */  /*: Bool */
```

Error

Control of the gradual type

```
let
  cast = x: x;
in
(from_gradual (cast (to_gradual 1)))
  /*# no-gradual */  /*: Bool */
```

Typechecks

And that's all for today...

POC implementation in OCaml

https://github.com/regnat/tix

(Very wip) rewrite in Haskell

https://github.com/regnat/ptyx

And that's all for today...

POC implementation in OCaml

https://github.com/regnat/tix

(Very wip) rewrite in Haskell

https://github.com/regnat/ptyx

regnat regnat@freenode ✉ regnat@regnat.ovh