

**PSC 2023/24 (375AA, 9CFU)**

## Principles for Software Composition

Roberto Bruni

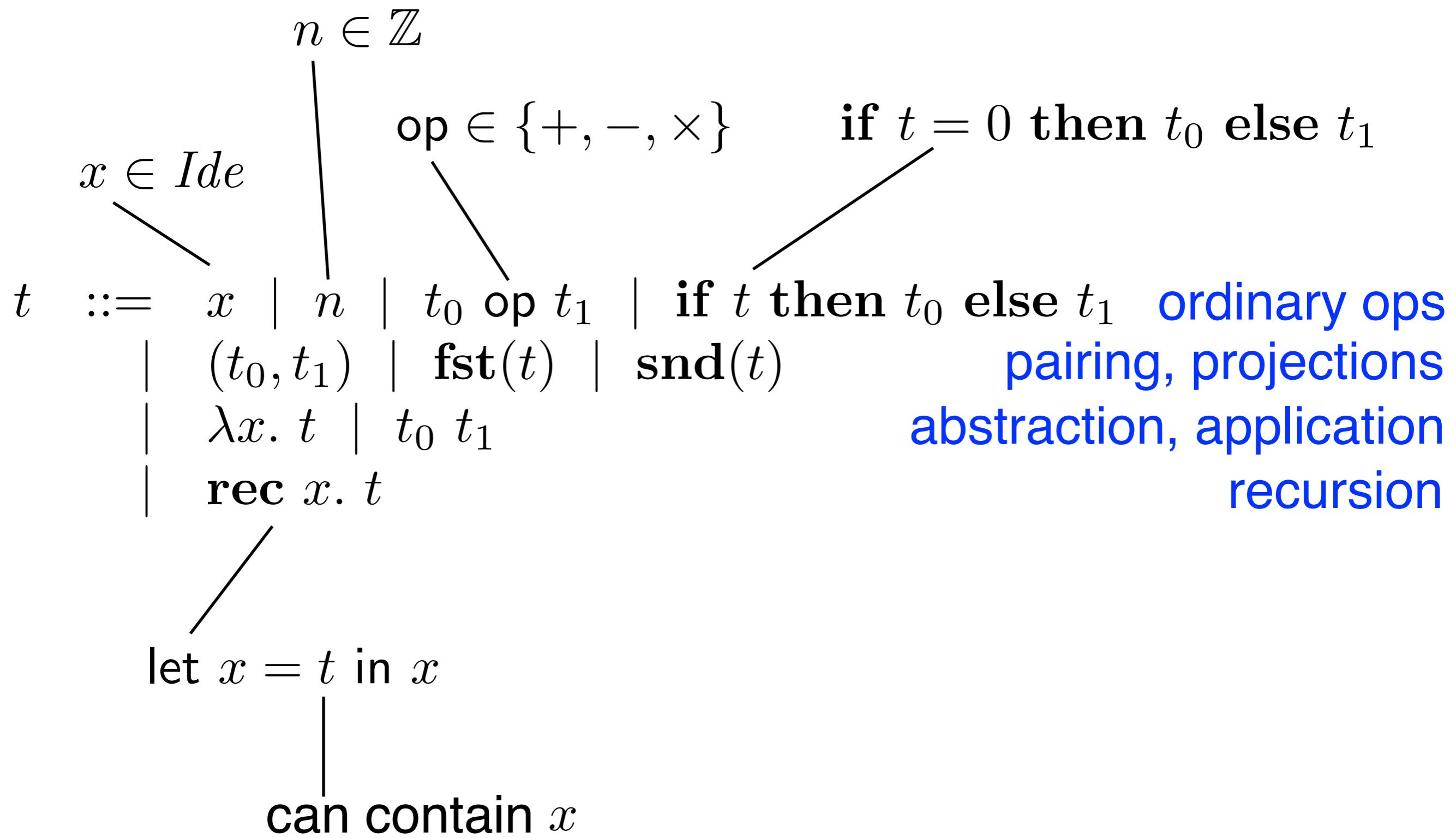
<http://www.di.unipi.it/~bruni/>

<http://didawiki.di.unipi.it/doku.php/magistraleinformatica/psc/start>

**12a - HOFL Syntax & Types**

# HOFL pre-terms (Higher Order Functional Language)

# HOFL Syntax





# Exercise

**rec**  $f.$   $\lambda x.$  **if**  $x$  **then** 1 **else**  $x \times (f\ (x - 1))$

guess the meaning of the above pre-term

factorial



# Exercise

```
rec rep. λn. λf. λx. if n then x  
                      else f (rep (n - 1) f x)
```

guess the meaning of the above pre-term

$$rep\ n\ f\ x = f^n\ x$$



# Exercise

$$\lambda x. \left( \left( \begin{array}{l} \text{rec } f. \lambda y. \text{ if } (x - y) \text{ then } 0 \\ \quad \text{else if } (x + y) \text{ then } 1 \\ \quad \text{else } f(y + 1) \end{array} \right) 0 \right)$$

guess the meaning of the above pre-term

greater or equal than 0

# Badge exercise



assuming  $true = 0$   
 $false = \text{any } n \neq 0$

fill the dots (in HOFL)

*or*  $\triangleq \lambda n. \lambda m. \dots$

*and*  $\triangleq \lambda n. \lambda m. \dots$

*not*  $\triangleq \lambda m. \dots$

*implies*  $\triangleq \lambda n. \lambda m. \dots$

*iff*  $\triangleq \lambda n. \lambda m. \dots$

# Pre-terms

```
 $t ::= x \mid n \mid t_0 \text{ op } t_1 \mid \text{if } t \text{ then } t_0 \text{ else } t_1$   
|  $(t_0, t_1) \mid \text{fst}(t) \mid \text{snd}(t)$   
|  $\lambda x. t \mid t_0 \ t_1$   
| rec  $x. t$ 
```

they are called pre-terms, why?

$x + 1$

$1 + (0, 5)$

**if**  $x$  **then**  $x + 1$  **else**  $x - 1$

$2 \times \lambda x. x$

$(0, \lambda x. x)$

**we need a  
type system**

$3 \lambda x. x + 1$

**fst** $(0, \lambda x. x)$

$\text{fst}(3)$

$(\lambda x. x + 1) \ 3$

**if**  $x$  **then**  $\lambda x. x$  **else**  $(x, x)$

**rec**  $f. \lambda x. x + (f \ 0)$

**rec**  $f. \lambda x. f + x$

# HOFL types

# Types

$$t ::= x \mid n \mid t_0 \text{ op } t_1 \mid \text{if } t \text{ then } t_0 \text{ else } t_1 \\ \mid (t_0, t_1) \mid \text{fst}(t) \mid \text{snd}(t) \\ \mid \lambda x. t \mid t_0 \ t_1 \\ \mid \text{rec } x. t$$

which types?

infinitely many combinations!

pairs

functions

*int*

*int \* int*

*int → int*

*int \* (int → int)*

*(int \* int) → int*

*int \* (int \* int)*

*(int → int) → int*

*(int → int) \* (int → int)*

*(int → int) → (int → (int \* int))*

# Types Syntax

$$\tau ::= \text{int} \mid \tau_0 * \tau_1 \mid \tau_0 \rightarrow \tau_1$$
$$\mathcal{T}$$
 set of all types

why not lists?

for the same reason we avoid division  
head and tail are not total functions

assume variables are typed

$$Ide = \{Ide_\tau\}_{\tau \in \mathcal{T}}$$
$$\widehat{\cdot}: Ide \rightarrow \mathcal{T}$$
$$\widehat{x}$$
 denotes the type of  $x$

# Type judgements

formulas:  $t : \tau$   reads “has type”

types are assigned to pre-terms  
using a set of inference rules  
(structural induction of HOFL syntax)

# Type system

$$\frac{}{x : \widehat{x}}$$
     $\frac{}{n : int}$      $\frac{t_0 : int \quad t_1 : int}{t_0 \text{ op } t_1 : int}$      $\frac{t : int \quad t_0 : \tau \quad t_1 : \tau}{\mathbf{if } \ t \ \mathbf{then} \ t_0 \ \mathbf{else} \ t_1 : \tau}$   
  
 $\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1}$      $\frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0}$      $\frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$   
  
 $\frac{x : \tau_0 \quad t : \tau_1}{\lambda x. \ t : \tau_0 \rightarrow \tau_1}$      $\frac{t_1 : \tau_0 \rightarrow \tau_1 \quad t_0 : \tau_0}{t_1 \ t_0 : \tau_1}$   
  
 $\frac{x : \tau \quad t : \tau}{\mathbf{rec} \ x. \ t : \tau}$

# Well-formed terms

$$\begin{aligned} t ::= & \quad x \mid n \mid t_0 \text{ op } t_1 \mid \text{if } t \text{ then } t_0 \text{ else } t_1 \\ & \mid (t_0, t_1) \mid \text{fst}(t) \mid \text{snd}(t) \\ & \mid \lambda x. t \mid t_0 t_1 \\ & \mid \mathbf{rec} \ x. t \end{aligned}$$
$$\tau ::= \text{int} \mid \tau_0 * \tau_1 \mid \tau_0 \rightarrow \tau_1$$
 $\mathcal{T}$  set of all types

a pre-term  $t$  is *well formed* if  $\exists \tau \in \mathcal{T}. t : \tau$

i.e. if we can assign a type to it  
also called *well-typed* or *typeable*

 $T_\tau$  set of all well-formed terms of type  $\tau$

# Type checking

# Example

variables are tagged with (declared) types

we deduce the type of terms by structural recursion

$$fact \triangleq \mathbf{rec} \ f : int \rightarrow int. \ \lambda x : int. \ \mathbf{if} \ x \ \mathbf{then} \ 1 \ \mathbf{else} \ x \times (f \ (x - 1))$$

---

$$fact : int \rightarrow int$$

# Example

variables are tagged with (declared) types

we deduce the type of terms by structural recursion

$$fact \triangleq \mathbf{rec} \ f : int \rightarrow int. \ \lambda x : int. \ \mathbf{if} \ x \ \mathbf{then} \ 1 \ \mathbf{else} \ x \times (f \ (x - 1))$$

$$f : int \rightarrow int \quad \lambda x. \ \mathbf{if} \ x \ \mathbf{then} \ 1 \ \mathbf{else} \ (x \times (f(x - 1))) : int \rightarrow int$$

---

$$fact : int \rightarrow int$$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

$$\frac{\widehat{f} = int \rightarrow int \quad x : int \quad \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int}{f : int \rightarrow int \quad \lambda x. \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \rightarrow int}$$

---

$$fact : int \rightarrow int$$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

$$\begin{array}{c} \widehat{x} = int \quad x : int \quad 1 : int \quad (x \times (f(x - 1))) : int \\ \hline \widehat{f} = int \rightarrow int \quad x : int \quad \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \\ \hline f : int \rightarrow int \quad \lambda x. \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \rightarrow int \\ \hline fact : int \rightarrow int \end{array}$$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

$$\begin{array}{c} \widehat{x} = int \\ \hline \widehat{x} = int \end{array} \quad \begin{array}{c} x : int \\ \hline x : int \end{array} \quad \begin{array}{c} 1 : int \\ \hline 1 : int \end{array} \quad \begin{array}{c} x : int \\ \hline x : int \end{array} \quad \begin{array}{c} f(x - 1) : int \\ \hline f(x - 1) : int \end{array}$$
$$\frac{\widehat{x} = int}{\widehat{f} = int \rightarrow int} \quad \frac{\widehat{x} = int \quad x : int \quad 1 : int}{x : int \quad \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int} \quad \frac{x : int \quad x : int \quad f(x - 1) : int}{(x \times (f(x - 1))) : int}$$
$$\frac{\widehat{f} = int \rightarrow int \quad x : int \quad \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int}{f : int \rightarrow int \quad \lambda x. \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \rightarrow int}$$

---

$$fact : int \rightarrow int$$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

$$\begin{array}{c} \widehat{x} = int \quad f : int \rightarrow int \quad x - 1 : int \\ \hline \widehat{x} = int \quad \quad \quad x : int \quad \quad \quad f(x - 1) : int \\ \hline \widehat{x} = int \quad x : int \quad 1 : int \quad \quad \quad (x \times (f(x - 1))) : int \\ \hline \widehat{f} = int \rightarrow int \quad \quad \quad \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \\ \hline f : int \rightarrow int \quad \quad \quad \lambda x. \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \rightarrow int \\ \hline \quad \quad \quad fact : int \rightarrow int \end{array}$$

# Example

variables are tagged with (declared) types  
 we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{ if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

$$\begin{array}{c}
 \frac{\widehat{f} = int \rightarrow int \quad x : int \quad 1 : int}{\widehat{x} = int \quad \frac{}{x : int} \quad \frac{}{f : int \rightarrow int} \quad \frac{}{x - 1 : int}}
 \\[10pt]
 \frac{\widehat{x} = int \quad \frac{}{x : int} \quad \frac{}{1 : int} \quad \frac{}{f(x - 1) : int}}{(x \times (f(x - 1))) : int}
 \\[10pt]
 \frac{\widehat{f} = int \rightarrow int \quad \frac{}{x : int} \quad \frac{}{\text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int}}{f : int \rightarrow int \quad \lambda x. \text{ if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \rightarrow int}
 \\[10pt]
 fact : int \rightarrow int
 \end{array}$$

# Example

variables are tagged with (declared) types  
 we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{ if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

$$\begin{array}{c}
 \widehat{x} = int \\
 \hline
 \widehat{f} = int \rightarrow int \\
 \hline
 \widehat{x} = int \quad \widehat{f} = int \rightarrow int \quad \widehat{x} = int \\
 \hline
 \frac{}{x : int} \quad \frac{}{f : int \rightarrow int} \quad \frac{}{x - 1 : int} \\
 \hline
 \widehat{x} = int \quad x : int \quad f(x - 1) : int \\
 \hline
 \widehat{x} = int \quad x : int \quad 1 : int \quad f(x - 1) : int \\
 \hline
 \widehat{x} = int \quad x : int \quad 1 : int \quad (x \times (f(x - 1))) : int \\
 \hline
 \widehat{f} = int \rightarrow int \quad x : int \quad \text{if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \\
 \hline
 f : int \rightarrow int \quad \lambda x. \text{ if } x \text{ then } 1 \text{ else } (x \times (f(x - 1))) : int \rightarrow int \\
 \hline
 fact : int \rightarrow int
 \end{array}$$

# Example

variables are tagged with (declared) types

we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \begin{matrix} f \\ \boxed{int} \end{matrix} . \lambda \begin{matrix} x \\ \boxed{int} \end{matrix} . \text{if } x \text{ then } 1 \text{ else } x \times (\begin{matrix} f \\ \boxed{int} \end{matrix} (x - 1))$

# Example

variables are tagged with (declared) types

we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \begin{matrix} f \\ \boxed{int \rightarrow int} \end{matrix} . \lambda \begin{matrix} x \\ \boxed{int} \end{matrix} . \text{if } \boxed{x} \text{ then } 1 \text{ else } x \times (\begin{matrix} f \\ \boxed{int} \end{matrix} (x - 1))$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \begin{matrix} f \\ int \rightarrow int \end{matrix} . \lambda \begin{matrix} x \\ int \end{matrix} . \text{if } \begin{matrix} x \\ int \end{matrix} \text{ then } \begin{matrix} 1 \\ int \end{matrix} \text{ else } x \times (\begin{matrix} f \\ int \end{matrix} (\begin{matrix} x - 1 \\ int \end{matrix}))$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \begin{matrix} f \\ int \rightarrow int \end{matrix} . \lambda \begin{matrix} x \\ int \end{matrix} . \text{if } \begin{matrix} x \\ int \end{matrix} \text{ then } \begin{matrix} 1 \\ int \end{matrix} \text{ else } \begin{matrix} x \\ int \end{matrix} \times (\begin{matrix} f \\ int \end{matrix} (\begin{matrix} x - 1 \\ int \end{matrix}))$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \begin{matrix} f \\ \boxed{int \rightarrow int} \end{matrix} . \lambda \begin{matrix} x \\ \boxed{int} \end{matrix} . \text{if } \begin{matrix} x \\ \boxed{int} \end{matrix} \text{ then } \begin{matrix} 1 \\ \boxed{int} \end{matrix} \text{ else } \begin{matrix} x \\ \boxed{int} \end{matrix} \times \left( \begin{matrix} f \\ \boxed{int \rightarrow int} \end{matrix} (\begin{matrix} x - 1 \end{matrix}) \right)$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \begin{matrix} f \\ int \rightarrow int \end{matrix} . \lambda \begin{matrix} x \\ int \end{matrix} . \text{if } \begin{matrix} x \\ int \end{matrix} \text{ then } \begin{matrix} 1 \\ int \end{matrix} \text{ else } \begin{matrix} x \\ int \end{matrix} \times (\begin{matrix} f \\ int \rightarrow int \end{matrix} (\begin{matrix} x \\ int \end{matrix} - 1))$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } f : \frac{}{int \rightarrow int} . \lambda x : \frac{}{int} . \text{if } \frac{}{int} x \text{ then } \frac{}{int} 1 \text{ else } \frac{}{int} x \times (\frac{}{int \rightarrow int} f : \frac{}{int} (\frac{}{int} x - \frac{}{int} 1))$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

more concisely

$$fact \stackrel{\text{def}}{=} \text{rec } \begin{array}{c} f \\ \boxed{int \rightarrow int} \end{array} . \lambda \begin{array}{c} x \\ \boxed{int} \end{array} . \text{if } \boxed{x} \text{ then } \boxed{1} \text{ else } \boxed{x} \times (\begin{array}{c} f \\ \boxed{int \rightarrow int} \end{array} \boxed{\begin{array}{c} x \\ \boxed{int} \end{array} - \boxed{1} \boxed{int}})$$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$$

more concisely

$$fact \stackrel{\text{def}}{=} \text{rec } \begin{array}{c} f \\ \boxed{int \rightarrow int} \end{array} . \lambda \begin{array}{c} x \\ \boxed{int} \end{array} . \text{if } \boxed{x} \text{ then } \boxed{1} \text{ else } \boxed{x} \times (\begin{array}{c} f \\ \boxed{int \rightarrow int} \end{array} \boxed{\begin{array}{c} x \\ \boxed{int} \end{array} - \boxed{1} \\ \boxed{int} \end{array})$$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } \underline{f} \ . \lambda \underline{x} : int. \text{if } \underline{x} \text{ then } \underline{1} \text{ else } \underline{x} \times (\underline{f}(\underline{x} - \underline{1}))$

The diagram illustrates the type derivation for the fact function. It shows the recursive definition  $fact \stackrel{\text{def}}{=} \text{rec } \underline{f} \ . \lambda \underline{x} : int. \text{if } \underline{x} \text{ then } \underline{1} \text{ else } \underline{x} \times (\underline{f}(\underline{x} - \underline{1}))$ . The type of  $\underline{f}$  is  $int \rightarrow int$ . The type of  $\underline{x}$  is  $int$ . The type of the entire term is  $int$ . The type of the if-then-else expression is  $int$ . The type of  $\underline{1}$  is  $int$ . The type of the multiplication expression is  $int$ . The type of the argument to  $\underline{f}$  is  $int$ . The type of the subtraction expression is  $int$ .

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } f : \boxed{int \rightarrow int} . \lambda \boxed{x : int} . \text{if } \boxed{x : int} \text{ then } \boxed{1 : int} \text{ else } \boxed{x : int} \times (\boxed{f : \boxed{int \rightarrow int} (\boxed{x : int} - \boxed{1 : int}) : int : int} : int)$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

```
fact  $\triangleq$  rec f : int  $\rightarrow$  int.  $\lambda x$  : int. if x then 1 else x  $\times$  (f (x - 1))
```

more concisely

$$fact \stackrel{\text{def}}{=} \text{rec } f \ . \lambda x. \text{if } x \text{ then } 1 \text{ else } x \times (\ f \ (x - 1))$$

$\boxed{int \rightarrow int}$        $\boxed{int}$        $\boxed{int}$        $\boxed{int}$        $\boxed{int}$        $\boxed{int \rightarrow int}$        $\boxed{int}$        $\boxed{int}$   
 $\boxed{int}$   
 $\boxed{int}$   
 $\boxed{int}$   
 $\boxed{int \rightarrow int}$

# Example

variables are tagged with (declared) types  
we deduce the type of terms by structural recursion

$fact \triangleq \text{rec } f : int \rightarrow int. \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1))$

more concisely

$fact \stackrel{\text{def}}{=} \text{rec } f \ . \lambda x : int. \text{if } x \text{ then } 1 \text{ else } x \times (f(x - 1)) : int \rightarrow int$

The diagram illustrates the type derivation for the `fact` function. It uses horizontal brackets to group parts of the term and vertical arrows to map them to their types. The innermost bracket groups the arguments of the multiplication:  $x$  and  $f(x - 1)$ . This is mapped to  $int$ . The next level groups the multiplication result and the recursive call:  $x \times (f(x - 1))$ . This is also mapped to  $int$ . The next level groups the if-then-else expression:  $1$ ,  $x$ , and the multiplication term. This is mapped to  $int$ . The outermost bracket groups the lambda abstraction and the if-then-else expression. This is mapped to  $int$ . The final level maps the entire term to  $int \rightarrow int$ .

# Type inference

# Example

types of variables are not given  
type rules are used to derive type constraints (type equations)  
whose solutions (via unification) define the principal type

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

intuitively

$$t \ 0 \equiv (0, (t \ 2)) \equiv (0, (2, (t \ 4))) \equiv \cdots \equiv (0, (2, (4, \ldots)))$$

sequence of all even numbers

we can type sequence of integers of fixed length  
we have no type for sequences of any/infinite length

# Example

types of variables are not given

type rules are used to derive type constraints (type equations)  
whose solutions (via unification) define the principal type

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \lambda x. (x, (p \ (x+2)))$$

Haskell

```
Prelude> let p x = (x, p (x+2))
```

```
<interactive>:...:5: error:
```

- Occurs check: cannot construct the infinite type:  $b \sim (t, b)$   
Expected type:  $t \rightarrow b$   
Actual type:  $t \rightarrow (t, b)$
- Relevant bindings include  
 $p :: t \rightarrow b$  (bound at <interactive>:...:5)

# Example

types of variables are not given  
type rules are used to derive type constraints (type equations)  
whose solutions (via unification) define the principal type

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x+2)))$$

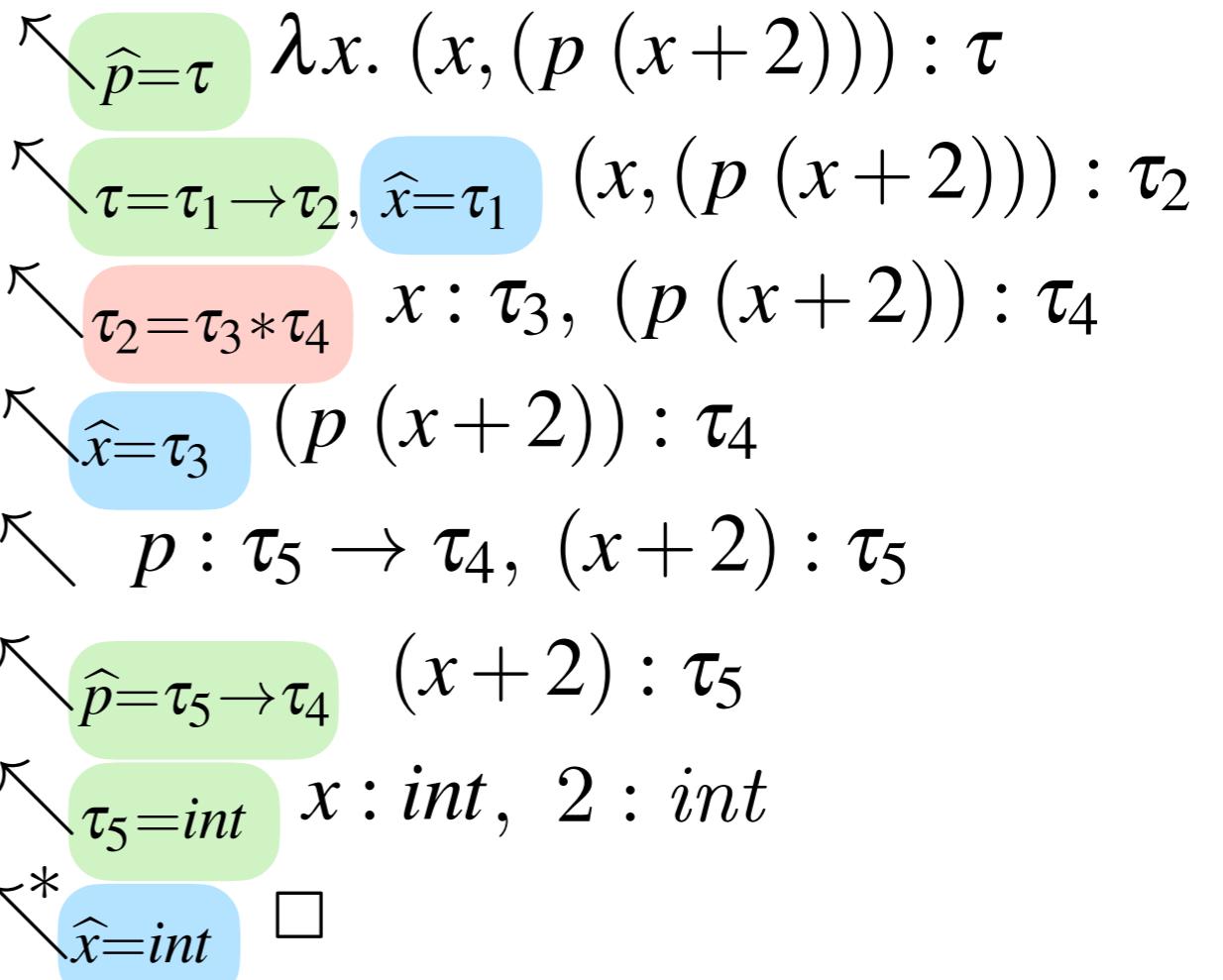
## Haskell

```
Prelude> let p x = x:(p (x+2))
p :: Num t => t -> [t]
Prelude> take 10 $ p 0
[0,2,4,6,8,10,12,14,16,18]
```

# Example

$t \stackrel{\text{def}}{=} \mathbf{rec}~p.~\lambda x.~(x, (p~(x + 2)))$

$t = \mathbf{rec}~p.\lambda x.(x,(p~(x+2))):\tau$



$$\left. \begin{array}{l} \widehat{x} = \tau_1 \\ \widehat{x} = \tau_3 \\ \widehat{x} = \text{int} \end{array} \right\} \quad \tau_1 = \tau_3 = \text{int}$$

$$\left. \begin{array}{l} \widehat{p} = \tau = \tau_1 \rightarrow \tau_2 \\ \widehat{p} = \tau_5 \rightarrow \tau_4 \end{array} \right\} \begin{array}{l} \tau_1 = \tau_5 = \text{int} \\ \tau_2 = \tau_4 \end{array}$$

$$\left. \begin{array}{l} \tau_2 = \tau_4 \\ \tau_2 = \tau_3 * \tau_4 \end{array} \right\} \text{fail! (occur check)}$$

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ p. \ \lambda \underline{x}. \ (\underline{x}, (p \underset{int}{\underline{\underline{}}} (x + 2)))$$

# Example

$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$

more concisely

$t = \mathbf{rec} \ p. \ \lambda \underline{x}. \ (\underline{x}, (p \ (\frac{x}{\boxed{int}} + \frac{2}{\boxed{int}})))$

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ p. \ \lambda \frac{x}{\mathbf{int}}. \ (\frac{x}{\mathbf{int}}, (\ p \ (\frac{x + 2}{\mathbf{int}})))$$

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ p. \ \lambda \begin{matrix} x \\ \text{int} \end{matrix}. \ (\begin{matrix} x \\ \text{int} \end{matrix}, (\begin{matrix} p \\ \text{int} \end{matrix} \ \underbrace{(\begin{matrix} x + 2 \\ \text{int} \end{matrix})}_{\text{int}}))$$

# Example

```
t =def rec p. λx. (x, (p (x + 2)))
```

more concisely

$$t = \mathbf{rec}~p.~\lambda_{\boxed{\mathbf{int}}} x_{\mathbf{int}}.~(x_{\boxed{\mathbf{int}}}, (\underbrace{p_{\mathbf{int} \rightarrow \tau_4}(x_{\boxed{\mathbf{int}}} + 2_{\mathbf{int}})}_{\mathbf{int}}))$$

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ \underbrace{p}_{int \rightarrow \tau_4}. \ \lambda \underbrace{x}_{int}. \ (\underbrace{x}_{int}, (\underbrace{p}_{int \rightarrow \tau_4} \ \underbrace{(\underbrace{x}_{int} + \underbrace{2}_{int})}_{int}))$$

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ \boxed{p} \ . \ \lambda \boxed{x} \ . \ (\boxed{x}, (\boxed{p} \ (\boxed{x} + \boxed{2})))$$

*int* *int* *int* *int* *int* *int*

*int*

*τ<sub>4</sub>*

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ \boxed{p} \ . \ \lambda \boxed{x} \ . \ (\boxed{x}, (\boxed{p} \ (\boxed{x} + \boxed{2})))$$

*int* *int* *int* *int* *int* *int*

*int*

*τ₄*

*int\*τ₄*

# Example

$$t \stackrel{\text{def}}{=} \mathbf{rec} \ p. \ \lambda x. \ (x, (p \ (x + 2)))$$

more concisely

$$t = \mathbf{rec} \ \boxed{p} \ . \ \lambda \boxed{x} \ . \ (\boxed{x}, (\boxed{p} \ (\boxed{x} + \boxed{2})))$$

*int* *int* *int* *int* *int* *int*

*int*

*τ₄*

*int\*τ₄*

*(int → (int\*τ₄)) = (int → τ₄) ⇒ τ₄ = (int\*τ₄)*

fail (occur check)



# Exercise

```
rec rep.  $\lambda n. \lambda f. \lambda x. \text{if } n \text{ then } x$   
                   $\text{else } f (\text{rep } (n - 1) f x)$ 
```

infer the type of the above term



# Exercise

$$\lambda x. \left( \left( \begin{array}{l} \text{rec } f. \lambda y. \text{ if } (x - y) \text{ then } 0 \\ \quad \text{else if } (x + y) \text{ then } 1 \\ \quad \text{else } f(y + 1) \end{array} \right) 0 \right)$$

infer the type of the above term

# Capture-avoiding substitutions (again)

# Free variables

$$\text{fv}(n) \stackrel{\text{def}}{=} \emptyset$$

$$\text{fv}(x) \stackrel{\text{def}}{=} \{x\}$$

$$\text{fv}(t_0 \text{ op } t_1) \stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1)$$

$$\text{fv}(\mathbf{if } t \text{ then } t_0 \text{ else } t_1) \stackrel{\text{def}}{=} \text{fv}(t) \cup \text{fv}(t_0) \cup \text{fv}(t_1)$$

$$\text{fv}((t_0, t_1)) \stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1)$$

$$\text{fv}(\mathbf{fst}(t)) \stackrel{\text{def}}{=} \text{fv}(t)$$

$$\text{fv}(\mathbf{snd}(t)) \stackrel{\text{def}}{=} \text{fv}(t)$$

$$\text{fv}(\lambda x. t) \stackrel{\text{def}}{=} \text{fv}(t) \setminus \{x\}$$

$$\text{fv}((t_0 \ t_1)) \stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1)$$

$$\text{fv}(\mathbf{rec } x. t) \stackrel{\text{def}}{=} \text{fv}(t) \setminus \{x\}$$

# Substitutions

$$n[t/x] = n$$

$$y[t/x] \stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(t_0 \text{ op } t_1)[t/x] \stackrel{\text{def}}{=} t_0[t/x] \text{ op } t_1[t/x] \quad \text{with op} \in \{+, -, \times\}$$

$$(\mathbf{if } t' \mathbf{ then } t_0 \mathbf{ else } t_1)[t/x] \stackrel{\text{def}}{=} \mathbf{if } t'[t/x] \mathbf{ then } t_0[t/x] \mathbf{ else } t_1[t/x]$$

$$(t_0, t_1)[t/x] \stackrel{\text{def}}{=} (t_0[t/x], t_1[t/x])$$

$$\mathbf{fst}(t')[t/x] \stackrel{\text{def}}{=} \mathbf{fst}(t'[t/x])$$

$$\mathbf{snd}(t')[t/x] \stackrel{\text{def}}{=} \mathbf{snd}(t'[t/x])$$

$$(t_0 \ t_1)[t/x] \stackrel{\text{def}}{=} (t_0[t/x] \ t_1[t/x])$$

$$(\lambda y. t')[t/x] \stackrel{\text{def}}{=} \lambda z. (t'[z/y][t/x]) \quad \text{for } z \notin \text{fv}(\lambda y. t') \cup \text{fv}(t) \cup \{x\}$$

$$(\mathbf{rec } y. t')[t/x] \stackrel{\text{def}}{=} \mathbf{rec } z. (t'[z/y][t/x]) \quad \text{for } z \notin \text{fv}(\mathbf{rec } y. t') \cup \text{fv}(t) \cup \{x\}$$

# Types are respected

$$\text{TH. } \begin{array}{c} x_0 : \tau_0 \\ t_0 : \tau_0 \end{array} \quad t : \tau \quad \Rightarrow \quad t[t_0/x_0] : \tau$$

proof omitted  
(by structural induction  
of the stronger assertion  $t[\tilde{t}/\tilde{x}] : \tau$ )