Uncovering the Unknown

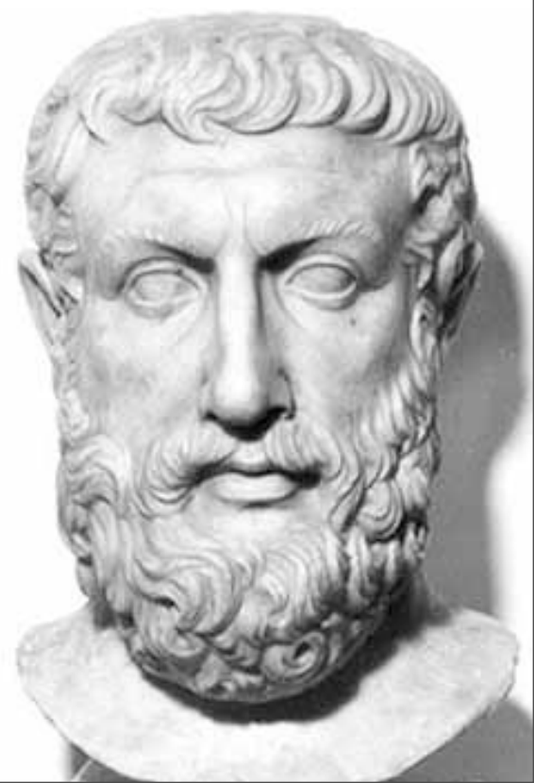
Principles of Type Inference

Agenda

- Philosophy of Types
- "Local" Type Inference
  - Scala
- "Global" Type Inference
  - SML
  - Haskell

Philosophy of Types

- Type system = Proof system
- Lots of little proofs about your program
- We can't detect *every* problem…
  - …but we can detect *some*!
- Not intended to make life difficult
  - (that happens by accident)



$$\frac{\Gamma \vdash t : T}{t \Rightarrow t' \ \vee \ t \text{ is value}} \textsc{Progress}$$

$$\frac{\Gamma \vdash t : T \qquad t \Rightarrow t'}{\Gamma \vdash t' : T} \textsc{Preservation}$$

Translation

Progress

- If a term is well-typed
- Then it evaluates
- *Or* it is already a value

Preservation

- If a term is well-typed
- *And* it evaluates
- Then the result has the *same* type

Example

```
name().length
```

Example

```
name().length
```

Has type: `Int`

Is value? `false`

Example

```
name().length
```

Has type: `Int`

Is value? `false`

Evaluates

```
"Daniel".length
```

Has type: `Int`

Is value? `false`

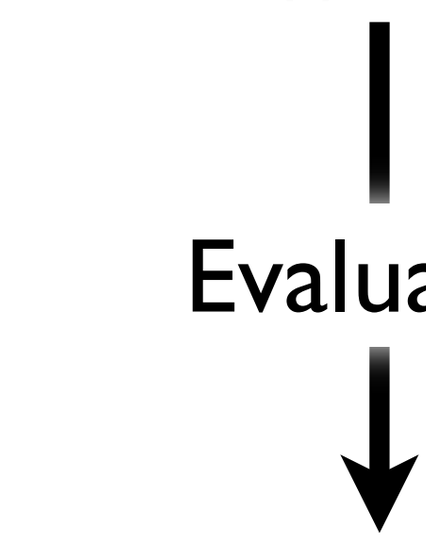
Example

```
name().length
```

Has type: `Int`

Is value? `false`

Evaluates



```
"Daniel".length
```

Has type: `Int`

Is value? `false`

Evaluates

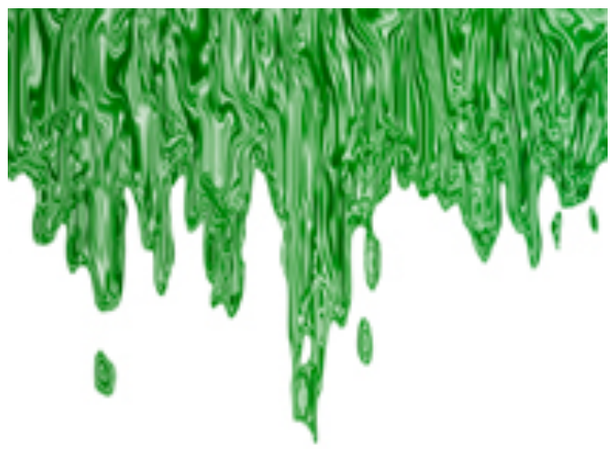
This is Type Theory!

Where do Types Come From?

Q: Why do languages make typing so explicit?

Q: Why do languages make typing so explicit?

A: **Laziness!**



Inference

- Type "inference" is a misnomer
  - (should be "reconstruction")
- Two possible approaches:
  - Each reconstruction is self-contained
  - Large "chunks" are considered holistically

Inference

- Type "inference" is a misnomer
  - (should be "reconstruction")
- ~~Two~~ One possible ~~approaches~~ approach:
  - Each reconstruction is self-contained
  - Large "chunks" are considered holistically

“Local” Inference

- Confine your focus to a single declaration
  - “Chunk” size is one statement
- Compute the type directly
- Move on to the next one…

```
def foo() = {
  val name = "Daniel"
  val len = name.length
  println(len)
}
```

```
def foo() = {
  val name = "Daniel"
  val len = name.length
  println(len)
}
```

```
def foo() = {
  val name = "Daniel":String
  val len = name.length
  println(len)
}
```

```
def foo() = {
  val name = "Daniel":String
  val len = name.length
  println(len)
}
```

```
def foo() = {
  val name = "Daniel":String
  val len = name.length :Int
  println(len)
}
```

```
def foo() = {
  val name = "Daniel" : String
  val len = name.length : Int
  println(len)
}
```

```
def foo() = {
  val name = "Daniel" : String
  val len = name.length : Int
  println(len) : Unit
}
```

: () ⇒ Unit

```
def foo() = {
  val name = "Daniel" : String
  val len = name.length : Int
  println(len) : Unit
}
```

```
val nums = List(1, 2, 3, 4, 5)
```

```
val strs = nums map { i => i.toString }
```

```
val nums = List(1, 2, 3, 4, 5)

val strs = nums map { i => i.toString }
```

: List[Int]

```
val nums = List(1, 2, 3, 4, 5)
```

```
val strs = nums map { i => i.toString }
```

: List[Int]

```
val nums = List(1, 2, 3, 4, 5)
```

```
val strs = nums map { i => i.toString }
```

: List[Int]

```
val nums = List(1, 2, 3, 4, 5)
```

```
val strs = nums map { i => i.toString }
```

: List[Int]

```
val nums = List(1, 2, 3, 4, 5)
```

: Int

```
val strs = nums map { i => i.toString }
```

: List[Int]

```
val nums = List(1, 2, 3, 4, 5)
```

: Int

```
val strs = nums map { i => i.toString }
```

: Int ⇒ String

: List[Int]

```
val nums = List(1, 2, 3, 4, 5)
```

: Int

```
val strs = nums map { i => i.toString }
```

: List[String]

: Int ⇒ String

Pros

- Mostly intuitive behavior

Pros

- Mostly intuitive behavior
- *Very* simple to implement
- (the compiler does this work anyway)

Pros

- Mostly intuitive behavior
- *Very* simple to implement
  - (the compiler does this work anyway)
- Always $O(1)$ and always decidable

Pros

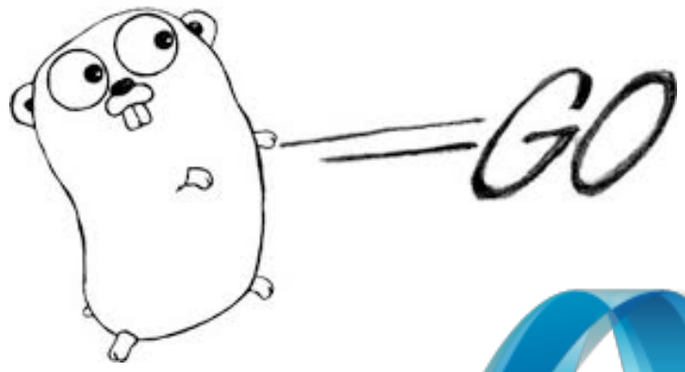
- Mostly intuitive behavior
- *Very* simple to implement
  - (the compiler does this work anyway)
- Always $O(1)$ and always decidable
- Quite beneficial in practice

Languages

Languages

Scala

Mirah



Microsoft® .NET

Java™

FANTOM

```
def add5(x) = x + 5
```


```
def add5(x) = x + 5
```

```
def fac(x: Int) = {
  if (x <= 1)
    1
  else
    x * fac(x - 1)
}
```

```
def fac(x: Int) = {
  if (x <= 1)
    1
  else
    x * fac(x - 1)
}
```

```
def fac(x: Int) = {
  if (x <= 1)
    1 : Int
  else
    x * fac(x - 1)
}
```

```
def fac(x: Int) = {
  if (x <= 1)
    1 : Int
  else
    x * fac(x - 1)
}
```

```
def fac(x: Int) = {
  if (x <= 1)
    1 : Int
  else
    x * fac(x - 1)
}
```

```
def fac(x: Int) = {
  if (x <= 1)
    1 : Int
  else
    x * fac(x - 1) : ????
}
```

“Global” Inference

- Doesn’t look at the *whole* program
- Simultaneously examines a large chunk
  - Usually a `let` binding (Hindley-Milner)
- Can be much smarter
- A generalization of local inference

```
def fac(x: Int) = {
  if (x <= 1)
    1
  else
    x * fac(x - 1)
}
```

```
fun fac (x: int) =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

```
fun fac (x: int) =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

$: int \to a$

```
fun fac (x: int) =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

: int → a

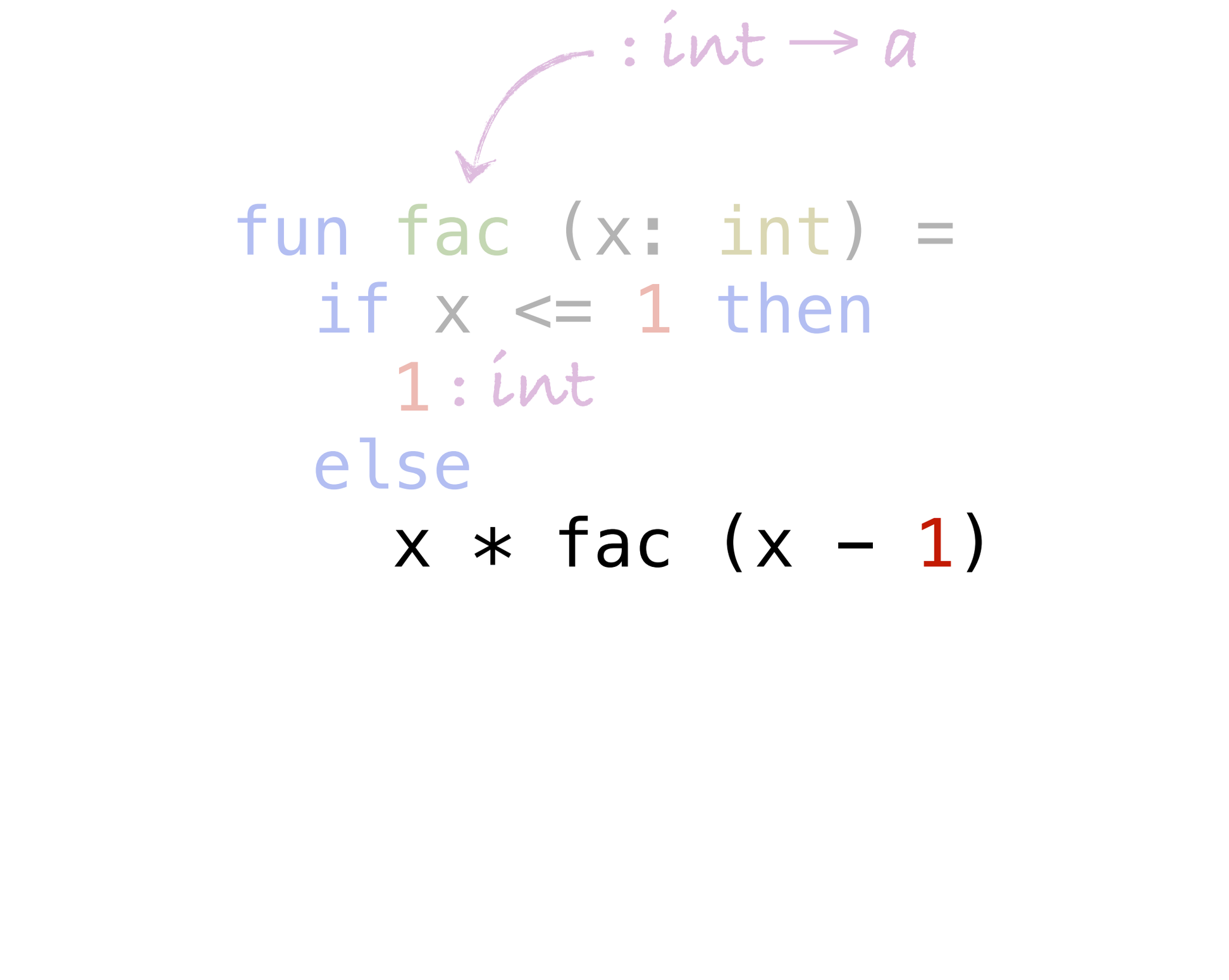
```
fun fac (x: int) =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

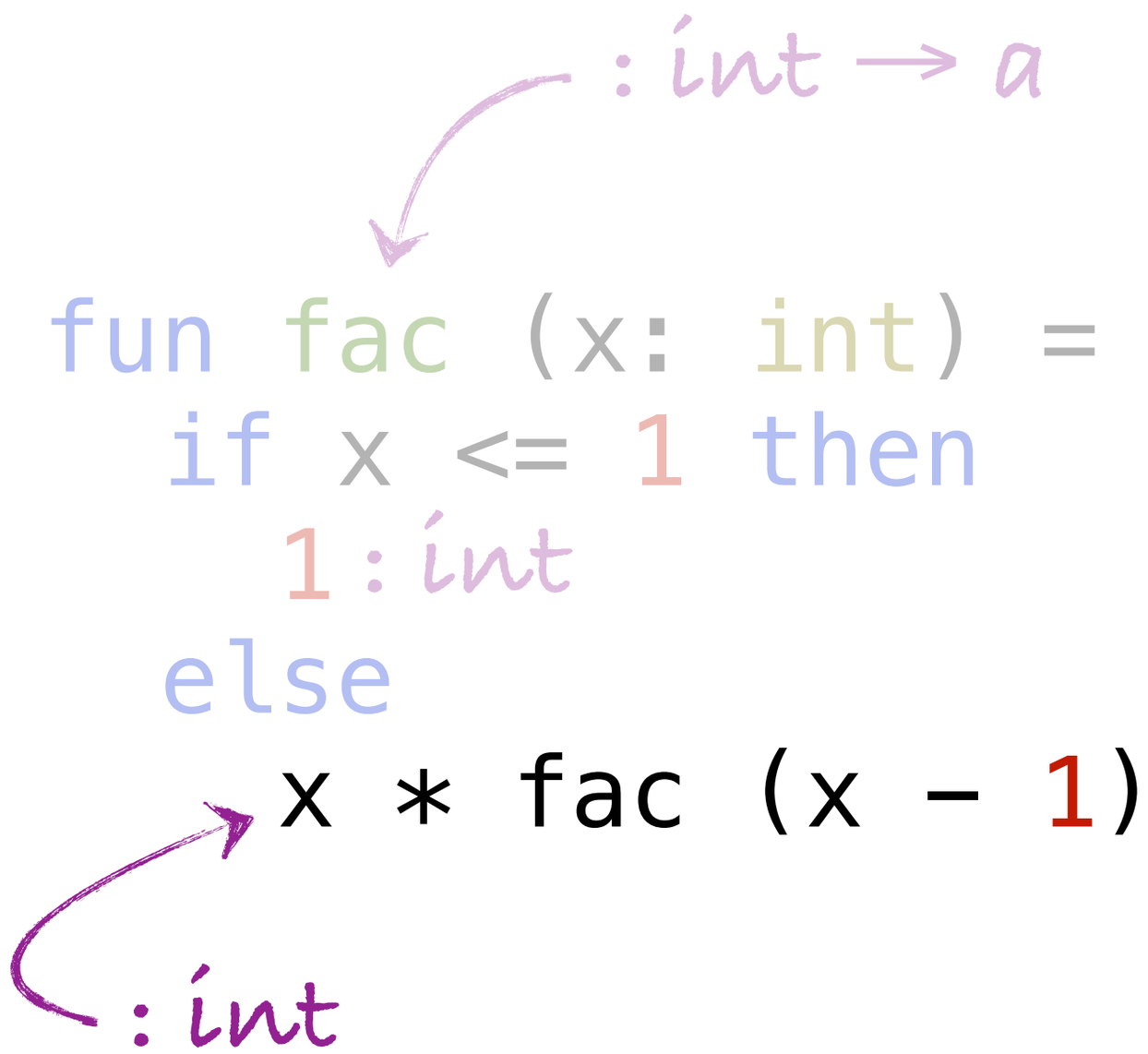
$: int \to a$

```
fun fac (x: int) =
  if x <= 1 then
    1 : int
  else
    x * fac (x - 1)
```

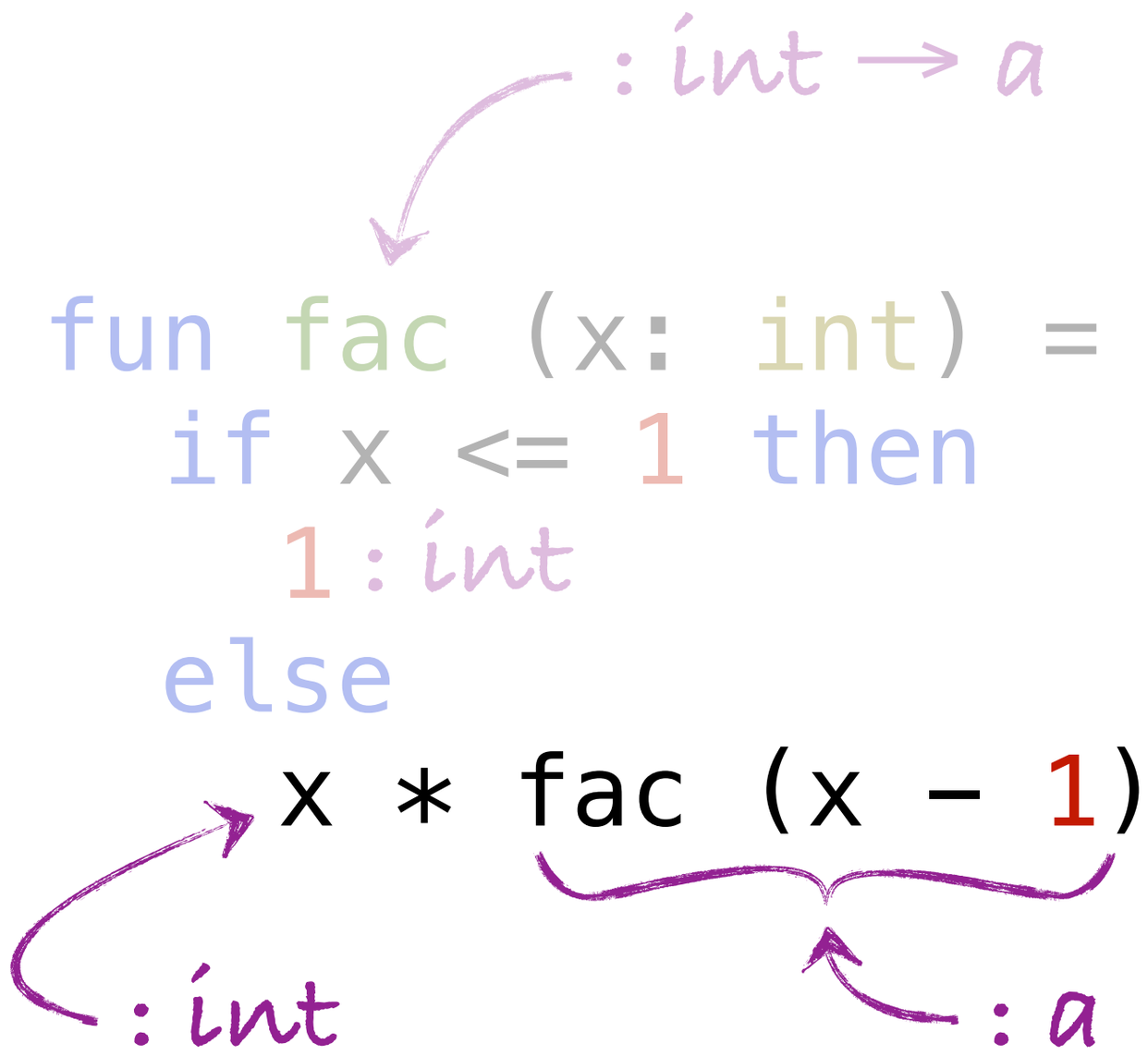
$\sigma = \{\}$



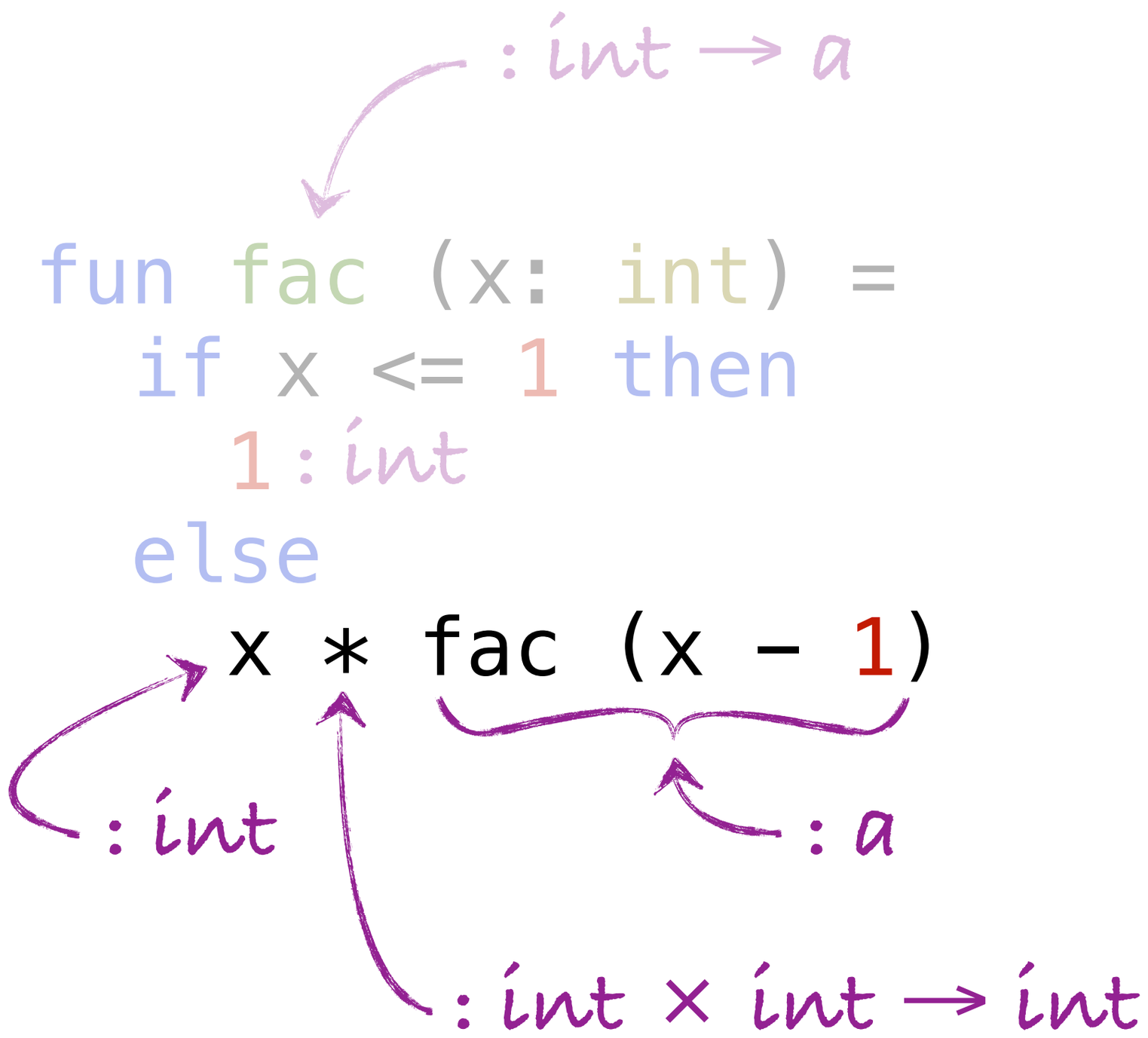
$\sigma = \{\}$



$\sigma = \{\}$



$\sigma = \{\}$



$\sigma = \{ a \mapsto int \}$

$: int \to a$

```
fun fac (x: int) =
  if x <= 1 then
    1 : int
  else
    x * fac (x - 1)
```

$: int$ $: a$

$: int \times int \to int$

: int → int

```
fun fac (x: int) =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

```
fun fac (x: int) =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

```
fun fac x =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

```
fun fac x =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

$: a \to b$

```
fun fac x =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$\sigma = \{\}$

$: a \to b$

```
fun fac x =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

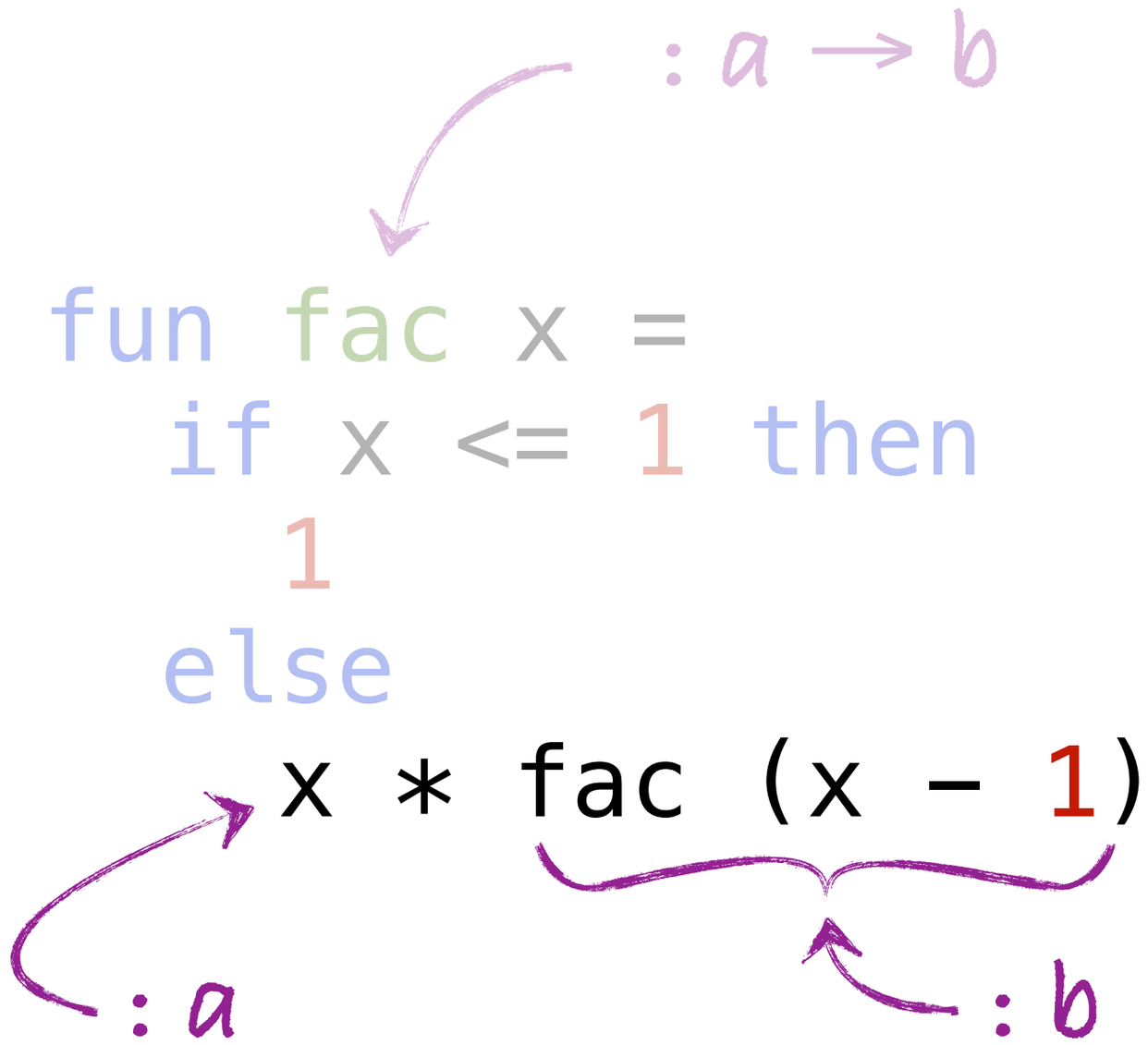
$\sigma = \{\}$

$: a \to b$

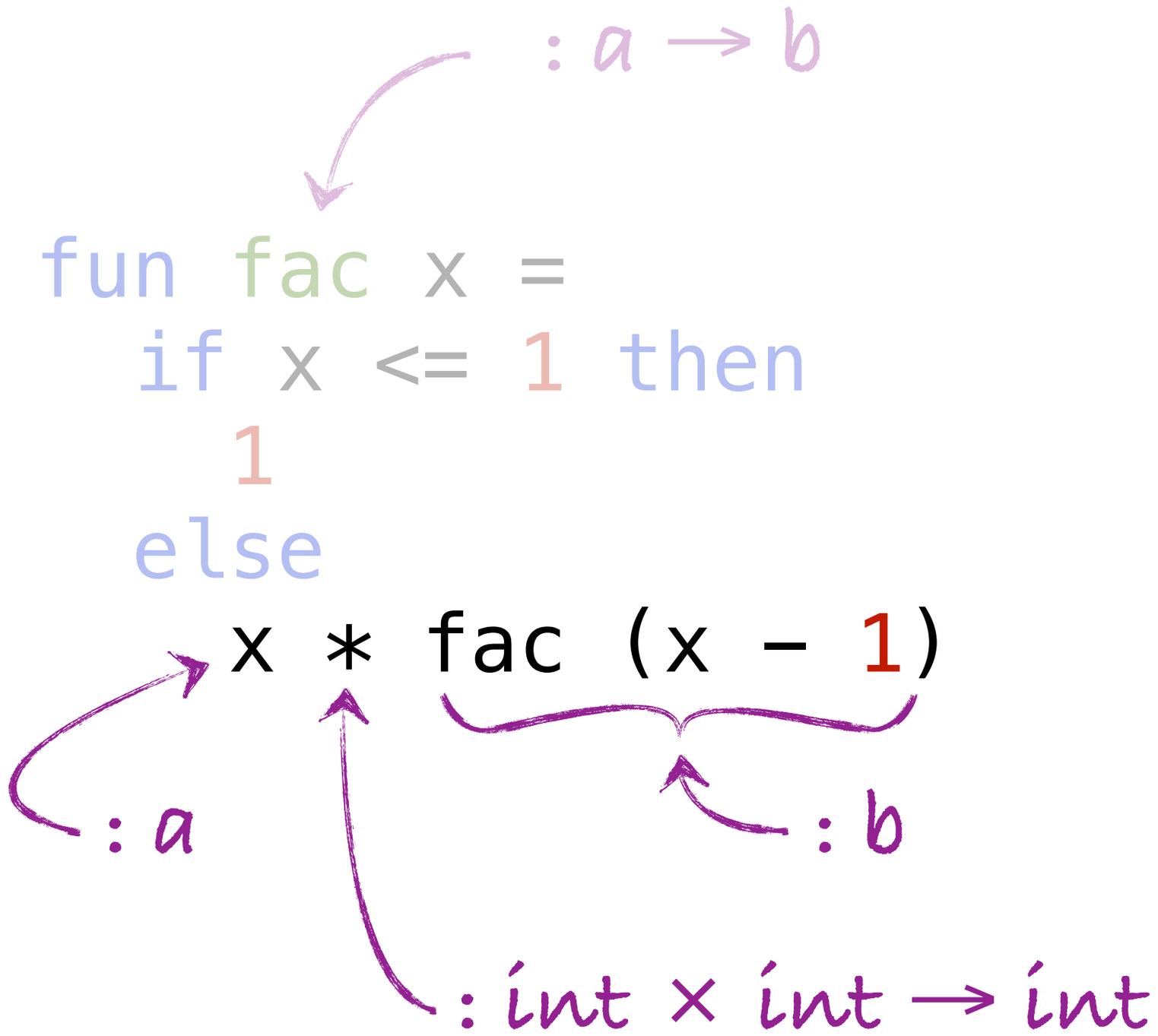
```
fun fac x =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

$: a$

$\sigma = \{\}$



$\sigma = \{\}$



$\sigma = \{a \mapsto int, b \mapsto int\}$

$: a \to b$

```
fun fac x =
  if x <= 1 then
     1
  else
     x * fac (x - 1)
```

$: a$ $: b$

$: int \times int \to int$

: *int* → *int*

```
fun fac x =
  if x <= 1 then
    1
  else
    x * fac (x - 1)
```

```
fun grow f x = (f x) :: x
```

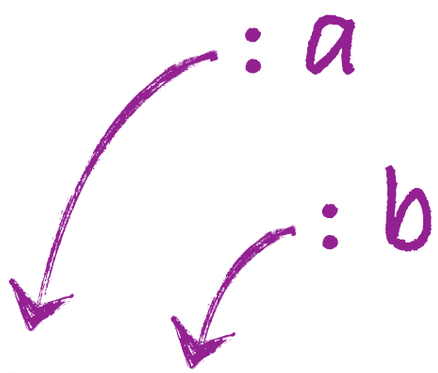
$\sigma = \{\}$

```
fun grow f x = (f x) :: x
```

$\sigma = \{\}$

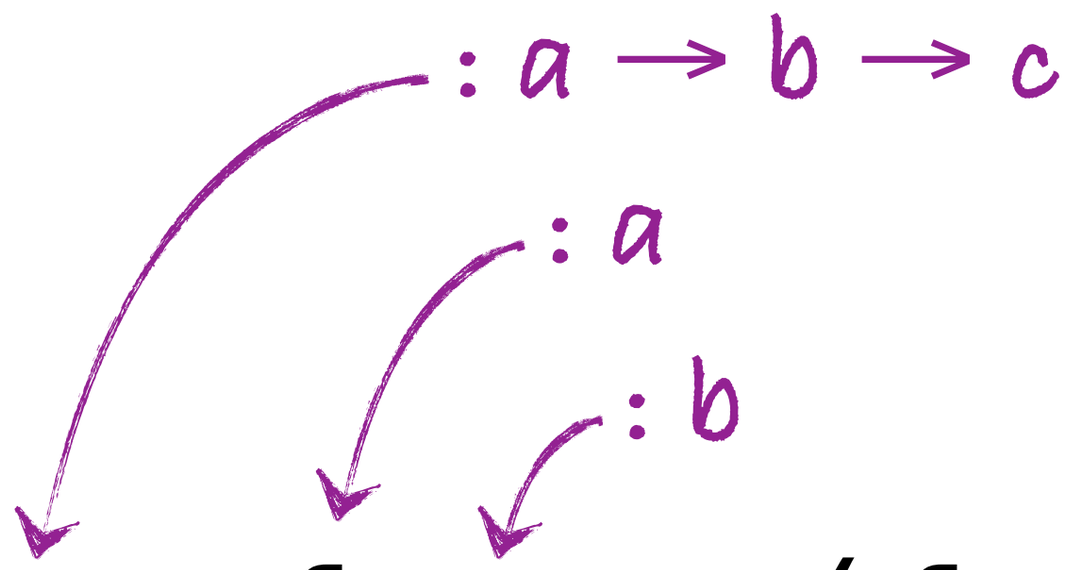

```
fun grow f x = (f x) :: x
```

$\sigma = \{\}$



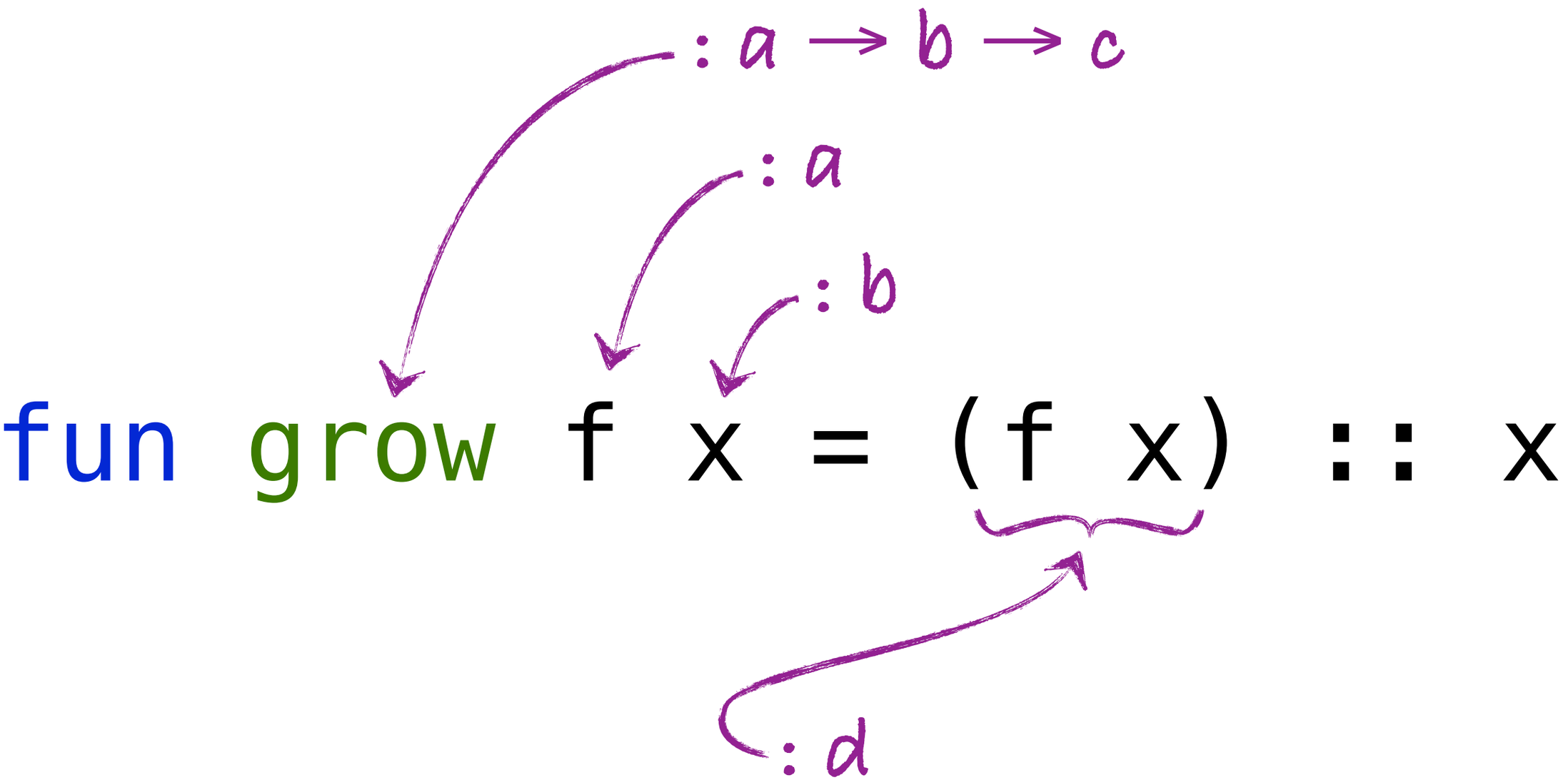

```
fun grow f x = (f x) :: x
```

$\sigma = \{\}$

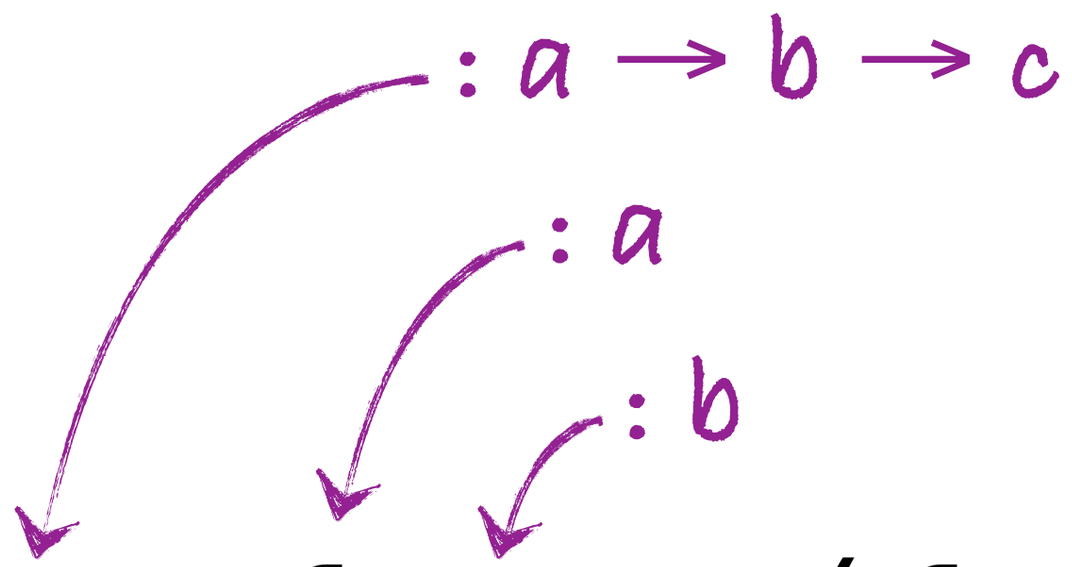



```
fun grow f x = (f x) :: x
```

$\sigma = \{\}$



$\sigma = \{a \mapsto (b \to d)\}$




```
fun grow f x = (f x) :: x
```

$\sigma = \{a \mapsto (b \to d)\}$

$: a \to b \to c$

$: a$

$: b$

```
fun grow f x = (f x) :: x
```

$: d$

$: d \times d\ \text{list} \to d\ \text{list}$

$\sigma = \{a \mapsto (b \to d),\ {}^{b}_{c} \mapsto (d\ \text{list})\}$

$: a \to b \to c$

$: a$

$: b$

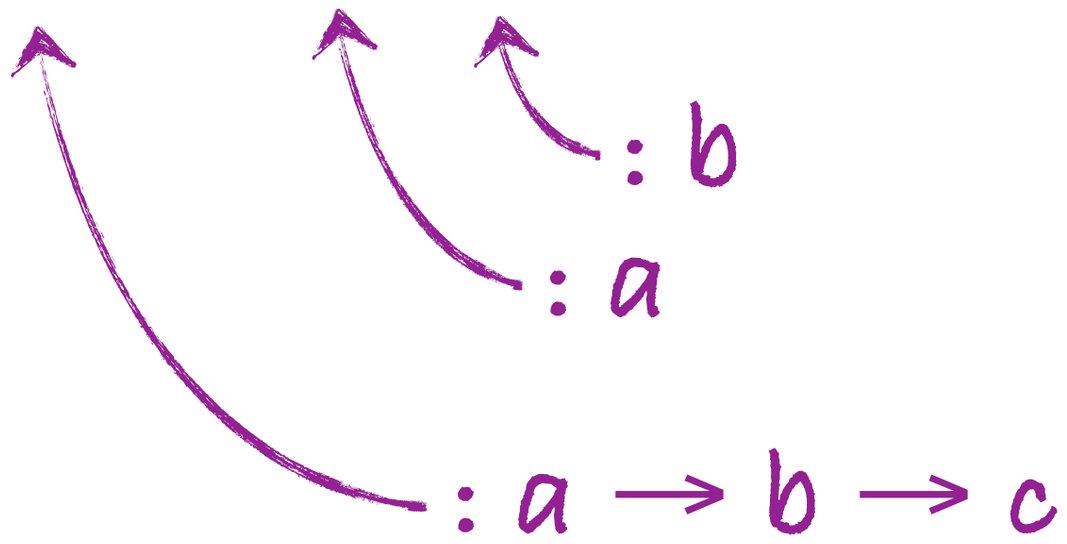
```
fun grow f x = (f x) :: x
```

$: d$

$: d \times d\ \text{list} \to d\ \text{list}$

$$\sigma = \{a \mapsto (b \to d),\ {}^{b}_{c} \mapsto (d\ \mathit{list})\}$$

```
fun grow f x = (f x) :: x
```

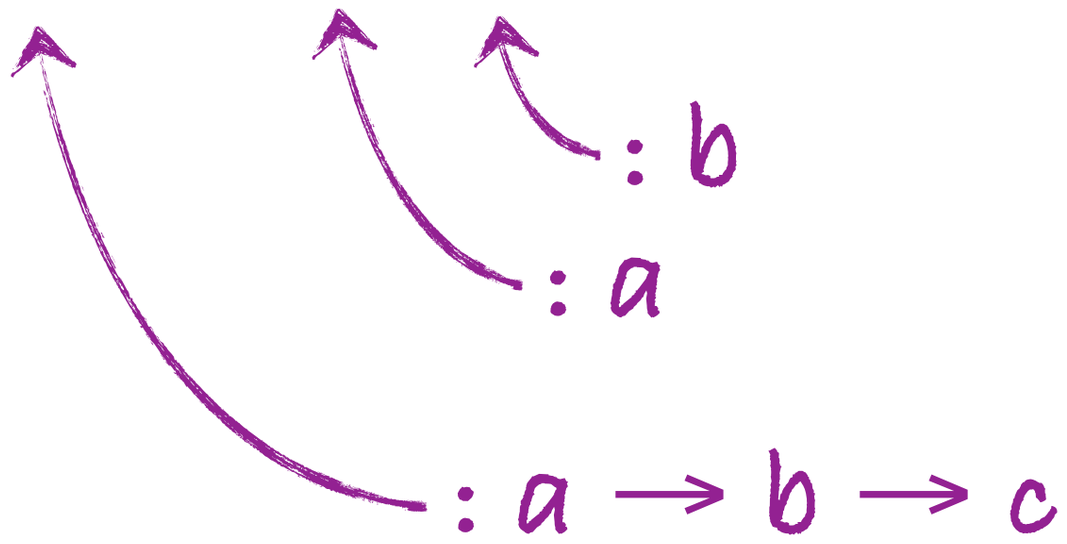


$$\sigma = \{a \mapsto (b \to d),\ \begin{matrix} b \\ c \end{matrix} \mapsto (d\ \mathit{list})\}$$

```
fun grow f x = (f x) :: x
```

$\sigma = \{a \mapsto (d\ \text{list} \to d),\ {}^{b}_{c} \mapsto (d\ \text{list})\}$

```
fun grow f x = (f x) :: x
```



: ('d list → 'd) → 'd list → 'd list

```
fun grow f x = (f x) :: x
```

```
def grow[D](f: List[D] => D)(x: List[D]): List[D] =
  f(x) :: x
```

Constraint Typing

- Hindley-Milner is a type system!
  - Damas-Milner is the algorithm

Constraint Typing

- Hindley-Milner is a type system!
  - Damas-Milner is the algorithm
- Damas-Milner gives us structure, not name

Constraint Typing

- Hindley-Milner is a type system!
  - Damas-Milner is the algorithm
- Damas-Milner gives us structure, not name
- Inherently *structural*, not *nominal*
  - …and that's why Scala doesn't have it

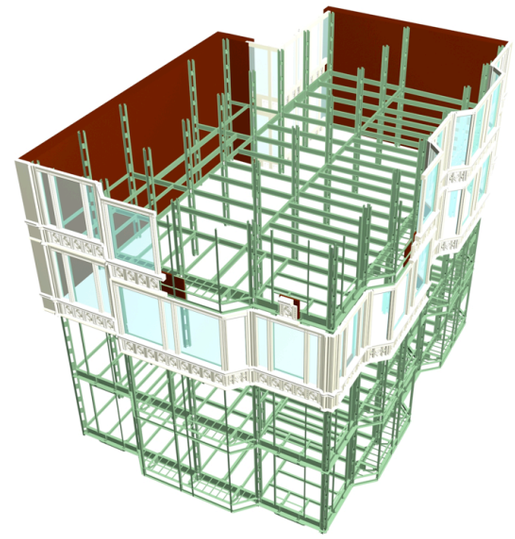
```
public interface Foo {
    public int length();
}

public interface Bar {
    public int length();
}


Foo f = …;
Bar b = f;          // really?
```

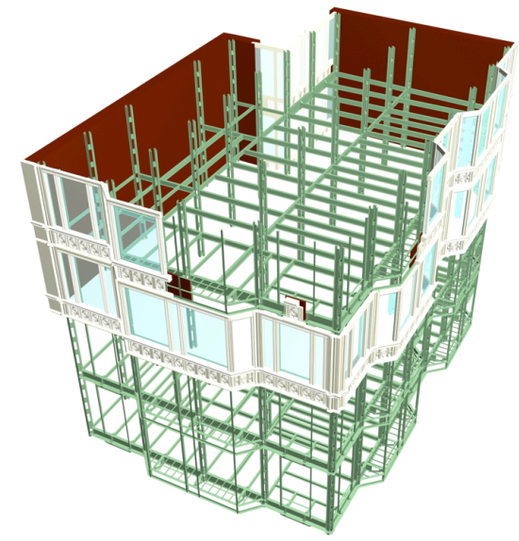
Structural Typing

- Compare types by what they *are*
  - (not by what they're *called*)



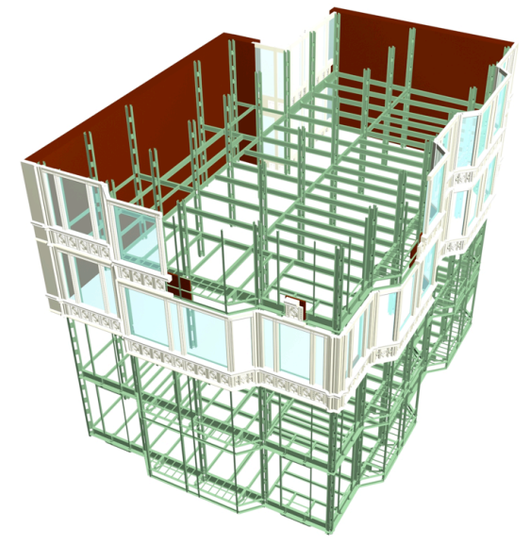
Structural Typing

- Compare types by what they *are*
  - (not by what they're *called*)
- This is exactly what "duck typing" means



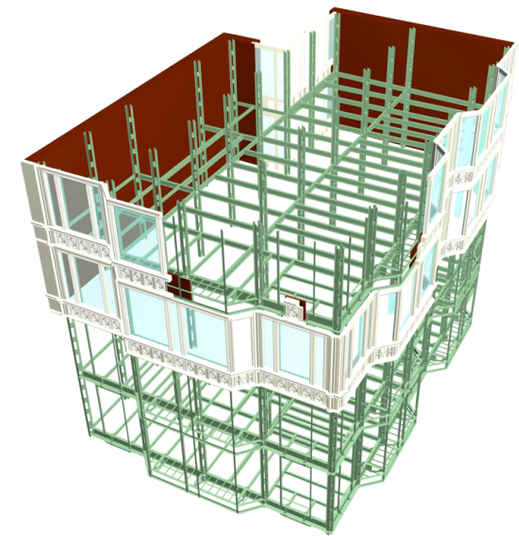
Structural Typing

- Compare types by what they *are*
  - (not by what they're *called*)
- This is exactly what "duck typing" means
- Really verbose error messages



Structural Typing

- Compare types by what they *are*
  - (not by what they're *called*)
- This is exactly what "duck typing" means
- Really verbose error messages
- Not Java Compatible



Statically typed languages of the future will use structural, rather than nominal, type systems.

Conclusion

- Type systems don't have to be horrible
  - (*a.k.a.* don't assume everything is Java)
- No excuse to not have *at least* local inference
- Inference can have a profound effect
- Nominal typing makes life…difficult

Conclusion

- **Type systems don't have to be horrible**
  - (*a.k.a.* don't assume everything is Java)
- No excuse to not have *at least* local inference
- Inference can have a profound effect
- Nominal typing makes life…difficult


Questions?