Calculus INT

Calculus INT

$$\textbf{Value Types} \quad A, B \ ::= \text{ types of the low-level language}$$

$$\textbf{Interactive Types} \quad X, Y \ ::= \ [A] \mid A \to X \mid X \multimap Y \mid A \cdot X \mid X \otimes Y \mid \forall \alpha \lhd A.\, X \mid \exists \alpha \lhd A.\, X$$

Related:

- Call-by-Push-Value [Levy 2004]
- Enrichted Effect Calculus [Møgelberg, Simpson 2010]
- Bounded Linear Logic [Girard, Scedrov, Scott 1992]
- Coeffect Calculi [Brunel, Gaboardi, Mazza, Zdancewic, 2014], [Orchard 2014], ...
- Tensorial Logic [Melliès 2012]

For simplicity, we discuss just the case with $\forall \alpha.\, X$ and $\exists \alpha.\, X$ here.

Calculus INT

Value Types A, B $::=$ types of the low-level language

Interactive Types X, Y $::=$ $[A] \mid A \to X \mid X \multimap Y \mid A \cdot X$
$\mid X \otimes Y \mid \forall \alpha \lhd A.\, X \mid \exists \alpha \lhd A.\, X$

Related:

- Call-by-Push-Value [Levy 2004]
- Enrichted Effect Calculus [Møgelberg, Simpson 2010]
- Bounded Linear Logic [Girard, Scedrov, Scott 1992]
- Coeffect Calculi [Brunel, Gaboardi, M
  [Orchard 2014], ...
- Tensorial Logic [Melliès 2012]

$$\texttt{unit} \cdot X \multimap X$$
$$(A \times B) \cdot X \multimap A \cdot (B \cdot X)$$

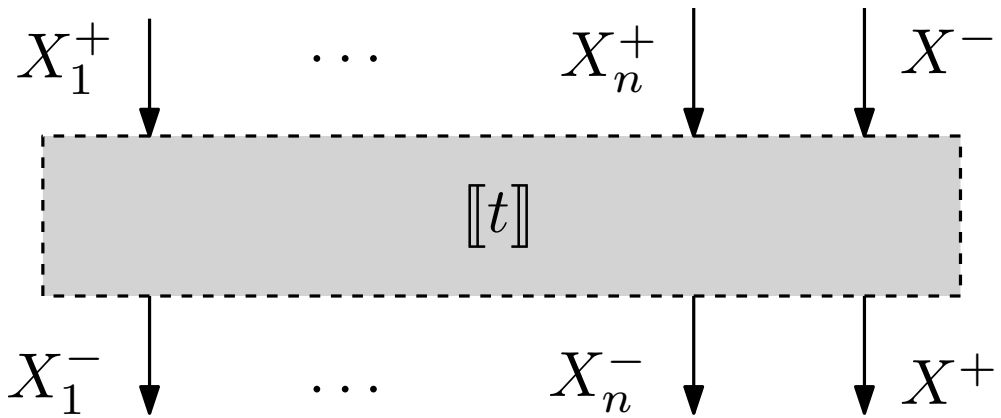
For simplicity, we discuss just the case with $\forall \alpha.\, X$ and $\exists \alpha.\, X$ here.

Type System

Typing Judgements

$$\Sigma \mid x_1{:}X_1, \ldots, x_n{:}X_n \vdash t\colon X$$

where Σ is a value context $y_1{:}B_1, \ldots, y_m{:}B_m$.



The value variables from Σ may occur freely in $[\![t]\!]$.

Typing Rules

Basic Computations

$$\frac{\Sigma \vdash v : A}{\Sigma \mid - \vdash \mathsf{return}\ v : [A]} \qquad \frac{\Sigma \mid \Gamma \vdash s : [A] \qquad \Sigma, x{:}A \mid \Delta \vdash t : [B]}{\Sigma \mid \Gamma, A \cdot \Delta \vdash \mathsf{let}\ x = s\ \mathsf{in}\ t : [B]}$$

Value functions

$$\frac{\Sigma, x{:}A \mid \Gamma \vdash t : X}{\Sigma \mid A \cdot \Gamma \vdash \mathsf{fn}\ x{:}A \Rightarrow t : A \to X} \qquad \frac{\Sigma \mid \Gamma \vdash s : A \to X \qquad \Sigma \vdash v : A}{\Sigma \mid \Gamma \vdash s(v) : X}$$

Typing Rules

Basic Computations

$$\frac{\Sigma \vdash v : A}{\Sigma \mid - \vdash \mathsf{return}\ v : [A]} \qquad \frac{\Sigma \mid \Gamma \vdash s : [A] \qquad \Sigma, x{:}A \mid \Delta \vdash t : [B]}{\Sigma \mid \Gamma, A \cdot \Delta \vdash \mathsf{let}\ x = s\ \mathsf{in}\ t : [B]}$$

Value functions

$$\frac{\Sigma, x{:}A \mid \Gamma \vdash t : X}{\Sigma \mid A \cdot \Gamma \vdash \mathsf{fn}\ x{:}A \Rightarrow t : A \to X} \qquad \frac{\Sigma \mid \Gamma \vdash s : A \to X \qquad \Sigma \vdash v : A}{\Sigma \mid \Gamma \vdash s(v) : X}$$

Typing Rules

Operations from the Low-Level Language

$$\frac{\Sigma \vdash v : A \times B \qquad \Sigma, x{:}A, y{:}B \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \texttt{let}\ (x, y) = v\ \texttt{in}\ t : X}$$

$$\frac{\Sigma \vdash v : \mu\alpha.\, A \qquad \Sigma, x{:}A[\mu\alpha.\, A/\alpha] \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \texttt{let}\ \texttt{fold}(x) = v\ \texttt{in}\ t : X}$$

$$\frac{\Sigma \vdash v : A + B \qquad \Sigma, x{:}A \mid \Gamma \vdash t_1 : X \qquad \Sigma, y{:}B \mid \Gamma \vdash t_2 : X}{\Sigma \mid \Gamma \vdash \texttt{case}\ v\ \texttt{of}\ \texttt{inl}(x) \to t_1\ ;\ \texttt{inr}(y) \to t_2 : X}$$

Primitive operations are constants, e.g. $\texttt{add} : \texttt{int} \times \texttt{int} \to [\texttt{int}]$.

Typing Rules

Operations from the Low-Level Language

$$\frac{\Sigma \vdash v : A \times B \qquad \Sigma, x{:}A, y{:}B \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \mathtt{let}\ (x, y) = v\ \mathtt{in}\ t : X}$$

$$\frac{\Sigma \vdash v : \mu\alpha.\, A \qquad \Sigma, x{:}A[\mu\alpha.\, A/\alpha] \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \mathtt{let}\ \mathtt{fold}(x) = v\ \mathtt{in}\ t : X}$$

$$\frac{\Sigma \vdash v : A + B \qquad \Sigma, x{:}A \mid \Gamma \vdash t_1 : X \qquad \Sigma, y{:}B \mid \Gamma \vdash t_2 : X}{\Sigma \mid \Gamma \vdash \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x) \to t_1\ ;\ \mathtt{inr}(y) \to t_2 : X}$$

Primitive operations are constants, e.g. $\mathsf{add} : \mathtt{int} \times \mathtt{int} \to [\mathtt{int}]$.

Typing Rules

Interactive Types

$$\frac{}{\Sigma \mid x{:}X \vdash x : X}$$

$$\frac{\Sigma \mid \Gamma, x{:}X \vdash t : Y}{\Sigma \mid \Gamma \vdash \lambda x{:}X.\, t : X \multimap Y} \qquad \frac{\Sigma \mid \Gamma \vdash s : X \multimap Y \qquad \Sigma \mid \Delta \vdash t : X}{\Sigma \mid \Gamma, \Delta \vdash s\ t : Y}$$

$$\frac{\Sigma \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \Lambda\alpha.\, t : \forall\alpha.\, X}\ \alpha \text{ not in } \Sigma, \Gamma \qquad \frac{\Sigma \mid \Gamma \vdash t : \forall\alpha.\, X}{\Sigma \mid \Gamma \vdash t\ A : X[A/\alpha]}$$

(rules for $\otimes$ and $\exists$ omitted)

Typing Rules

Interactive Types

$$\frac{}{\Sigma \mid x{:}X \vdash x : X}$$

$$\frac{\Sigma \mid \Gamma, x{:}X \vdash t : Y}{\Sigma \mid \Gamma \vdash \lambda x{:}X.\, t : X \multimap Y} \qquad \frac{\Sigma \mid \Gamma \vdash s : X \multimap Y \qquad \Sigma \mid \Delta \vdash t : X}{\Sigma \mid \Gamma, \Delta \vdash s\ t : Y}$$

$$\frac{\Sigma \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \Lambda\alpha.\, t : \forall\alpha.\, X}\ \alpha \text{ not in } \Sigma, \Gamma \qquad \frac{\Sigma \mid \Gamma \vdash t : \forall\alpha.\, X}{\Sigma \mid \Gamma \vdash t\ A : X[A/\alpha]}$$

(rules for $\otimes$ and $\exists$ omitted)

Typing Rules

Subexponentials

$$\frac{\Sigma \mid \Gamma \vdash t : X}{\Sigma \mid A \cdot \Gamma \vdash t : A \cdot X}$$

$$\frac{\Sigma \mid \Delta \vdash s : X \qquad \Sigma \mid \Gamma, x : A \cdot X, y : B \cdot X \vdash t : Y}{\Sigma \mid \Gamma, (A + B) \cdot \Delta \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t : Y}$$

$$\frac{\Sigma \mid \Gamma, x{:}X \vdash t : Y}{\Sigma \mid \Gamma, x : \mathtt{unit} \cdot X \vdash t : Y} \qquad \frac{\Sigma \mid \Gamma, x : A \cdot (B \cdot X) \vdash t : Y}{\Sigma \mid \Gamma, x : (A \times B) \cdot X \vdash t : Y}$$

$$\frac{\Sigma \mid \Gamma, x : B \cdot X \vdash t : Y}{\Sigma \mid \Gamma, x : A \cdot X \vdash t : Y}\ B \lhd A \qquad \frac{\Sigma \mid \Gamma \vdash t : A \cdot X}{\Sigma \mid \Gamma \vdash t : B \cdot X}\ B \lhd A$$

Typing Rules

Subexponentials

$$\frac{\Sigma \mid \Gamma \vdash t : X}{\Sigma \mid A \cdot \Gamma \vdash t : A \cdot X}$$

$$\frac{\Sigma \mid \Delta \vdash s : X \qquad \Sigma \mid \Gamma, x : A \cdot X, y : B \cdot X \vdash t : Y}{\Sigma \mid \Gamma, (A + B) \cdot \Delta \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t : Y}$$

$$\frac{\Sigma \mid \Gamma, x{:}X \vdash t : Y}{\Sigma \mid \Gamma, x : \mathtt{unit} \cdot X \vdash t : Y} \qquad \frac{\Sigma \mid \Gamma, x : A \cdot (B \cdot X) \vdash t : Y}{\Sigma \mid \Gamma, x : (A \times B) \cdot X \vdash t : Y}$$

$$\frac{\Sigma \mid \Gamma, x : B \cdot X \vdash t : Y}{\Sigma \mid \Gamma, x : A \cdot X \vdash t : Y}\ B \lhd A \qquad \frac{\Sigma \mid \Gamma \vdash t : A \cdot X}{\Sigma \mid \Gamma \vdash t : B \cdot X}\ B \lhd A$$

Examples

$\lambda x{:}[\texttt{int}].\ \texttt{let v = } x \texttt{ in}$
$\qquad\qquad \texttt{let w = add(v, v) in}$
$\qquad\qquad \texttt{return w}$
$\qquad : [\texttt{int}] \multimap [\texttt{int}]$

(add is a constant of type $\texttt{int} \times \texttt{int} \to [\texttt{int}]$)

$\lambda x{:}[\texttt{int}].\ \lambda y{:}(\texttt{int} \cdot [\texttt{int}]).\ \texttt{let v = } x \texttt{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let w = } y \texttt{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad \texttt{add(v, w)}$
$\qquad : [\texttt{int}] \multimap \texttt{int} \cdot [\texttt{int}] \multimap [\texttt{int}]$

Examples

$\lambda f{:}(\mathtt{unit} + \mathtt{int}) \cdot ([\mathtt{bool}] \multimap [\mathtt{int}]).$
$\quad$ copy f as f_1, f_2 in
$\quad$ let v = f_1 (return true) in
$\quad$ let w = f_2 (return false) in
$\quad$ return (v, w)
$\quad\quad : (\mathtt{unit} + \mathtt{int}) \cdot ([\mathtt{bool}] \multimap [\mathtt{int}]) \multimap [\mathtt{int} \times \mathtt{int}]$

Higher-Order Examples (Kierstead terms):

$$\lambda f.\, \mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\, (f_2\ (\lambda y.\, y)))$$
$$: (1 + \alpha) \cdot (\alpha \cdot (X \multimap X) \multimap X) \multimap X$$

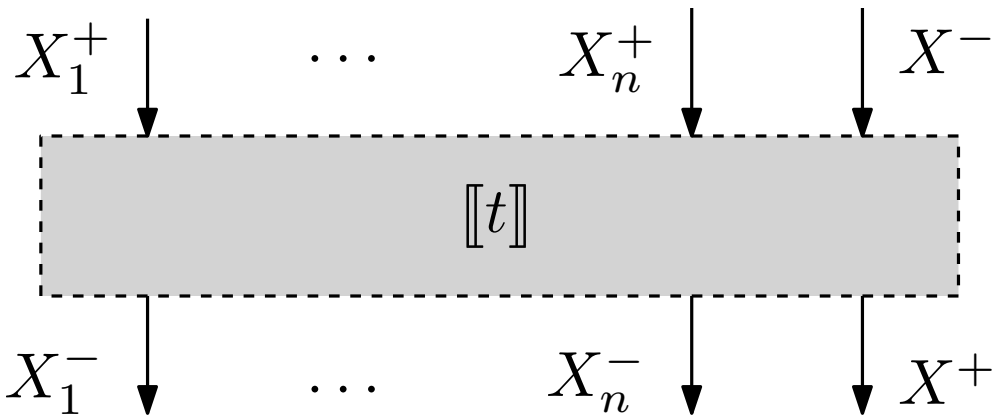
$$\lambda f.\, \mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\, (f_2\ (\lambda y.\, x)))$$
$$: (1 + \alpha) \cdot (\alpha \cdot (\alpha \cdot X \multimap X) \multimap X) \multimap X$$

Translation

A type judgement

$$\Sigma \mid x_1{:}X_1, \ldots, x_n{:}X_n \vdash t : X$$

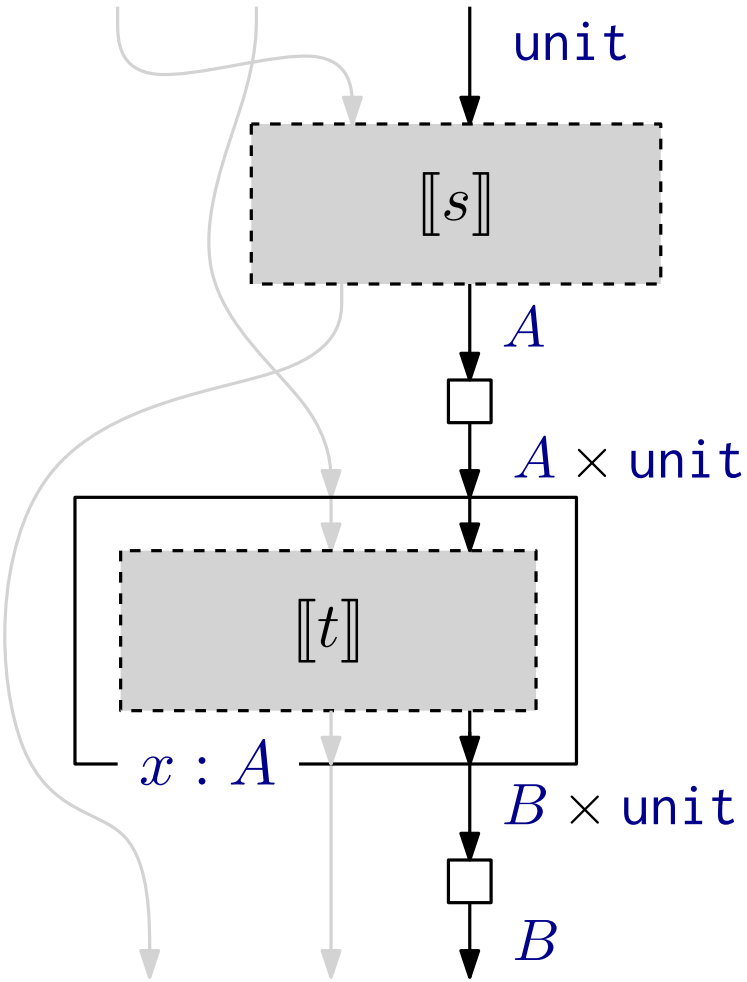
translates to a low-level program $[\![t]\!]$:



Translation

$$\frac{\Sigma \mid \Gamma \vdash s : [A] \qquad \Sigma, x{:}A \mid \Delta \vdash t : [B]}{\Sigma \mid \Gamma, A \cdot \Delta \vdash \mathtt{let}\ x = s\ \mathtt{in}\ t : [B]}$$

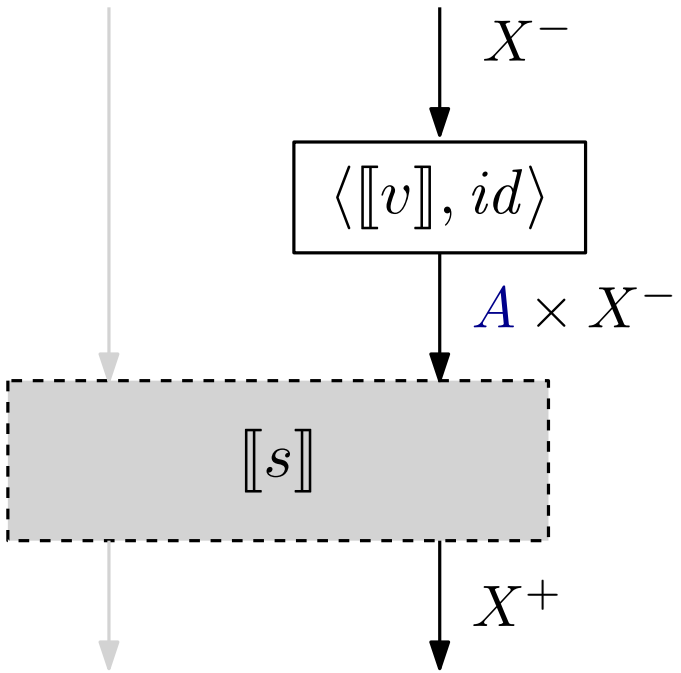
$[\![\mathtt{let}\ x = s\ \mathtt{in}\ t]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma \vdash s\colon A \to X \qquad \Sigma \vdash v\colon A}{\Sigma \mid \Gamma \vdash s(v)\colon X}$$

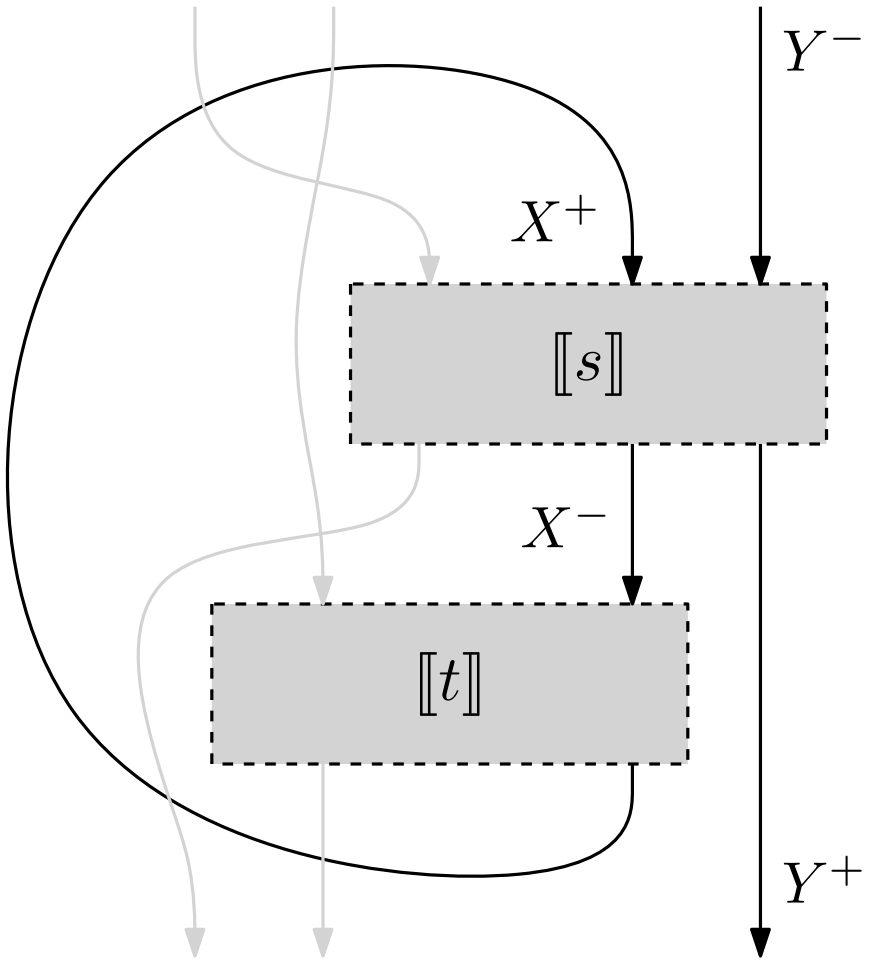
$[\![s(v)]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma \vdash s \colon X \multimap Y \qquad \Sigma \mid \Delta \vdash t \colon X}{\Sigma \mid \Gamma, \Delta \vdash s\ t \colon Y}$$

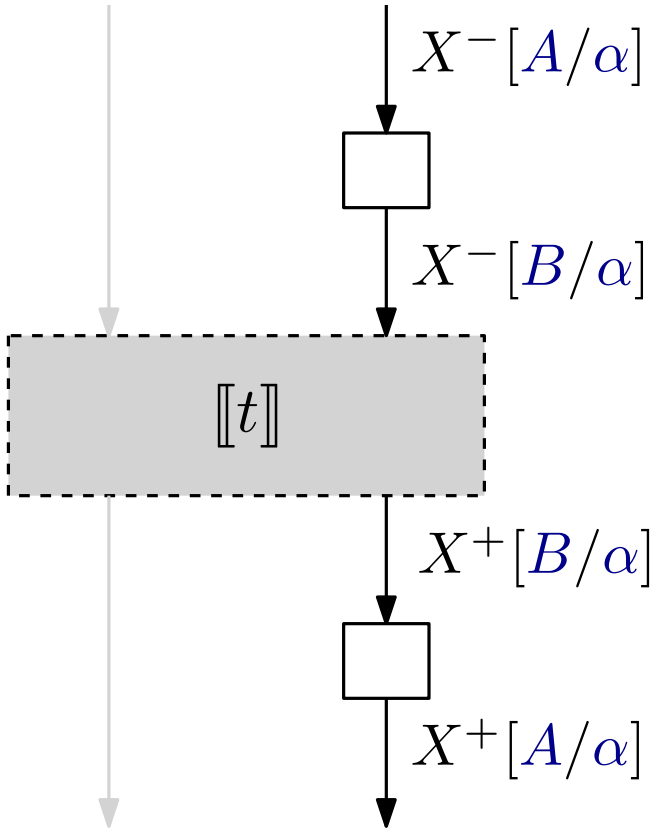
$[\![s\ t]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma \vdash t : \forall \alpha \lhd B.\, X}{\Sigma \mid \Gamma \vdash t\ A : X[A/\alpha]}\ A \lhd B$$

$[\![t\ A]\!] :=$



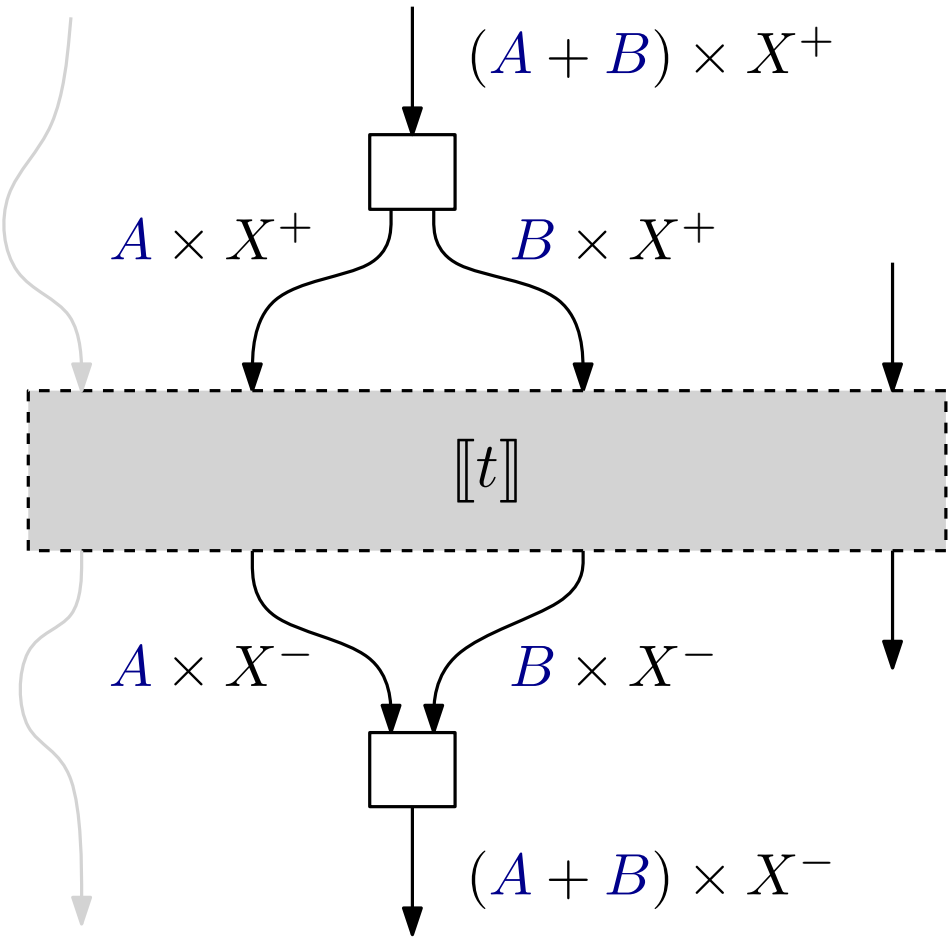
$\mathsf{enc}_{A,B} : A \to B$

$\mathsf{dec}_{A,B} : B \to A$

Translation

$$\frac{\Sigma \mid \Gamma, x\colon A \cdot X, y\colon B \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, z\colon (A + B) \cdot X \vdash \mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

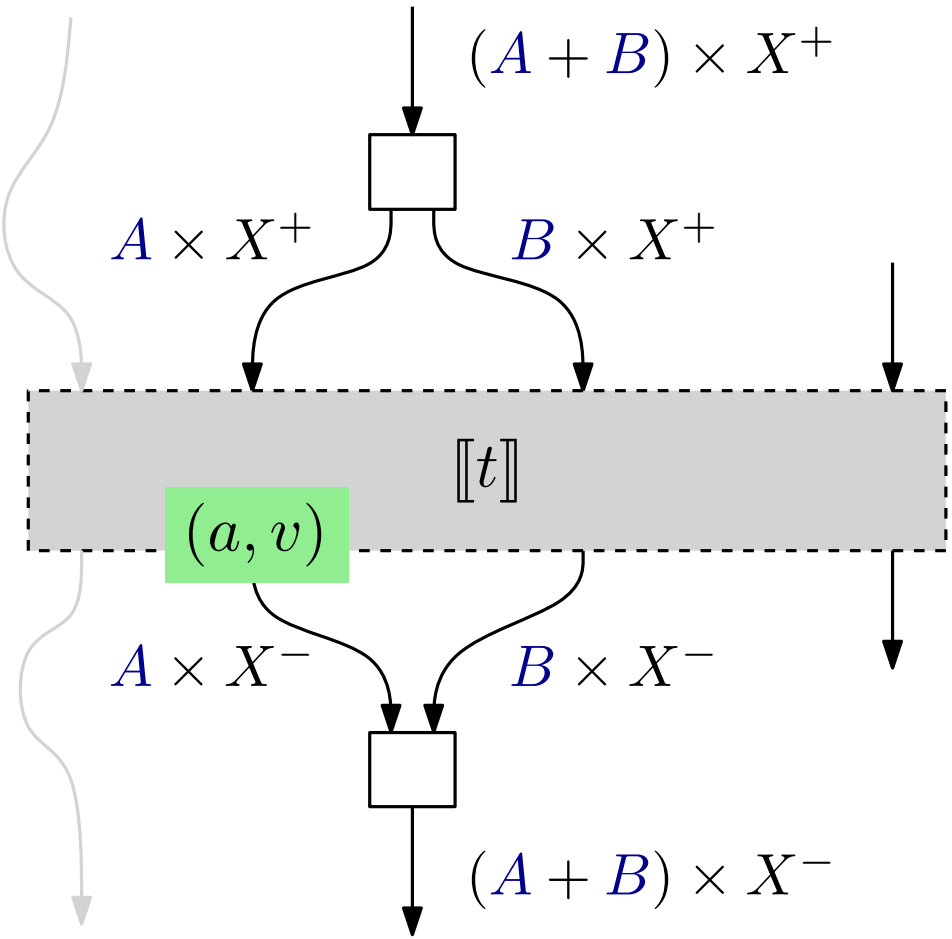
$[\![\mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma, x\colon A \cdot X, y\colon B \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, z\colon (A + B) \cdot X \vdash \mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

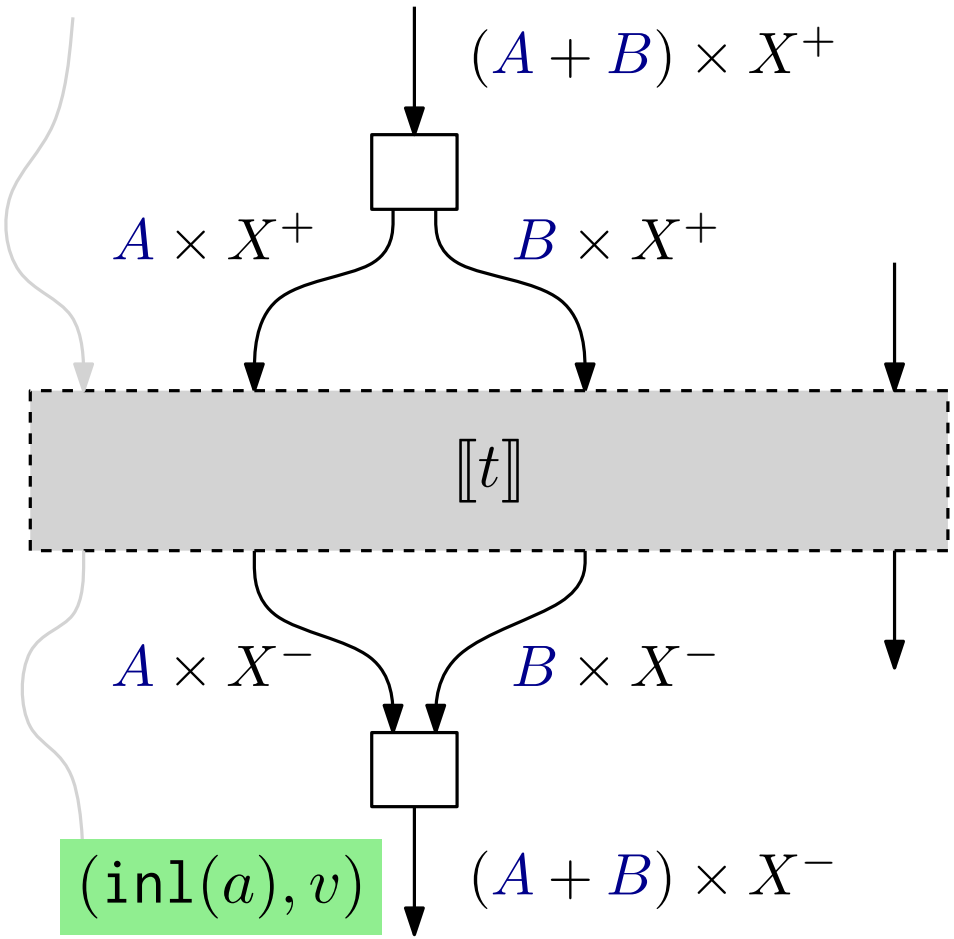
$[\![\mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma, x\colon A \cdot X, y\colon B \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, z\colon (A + B) \cdot X \vdash \mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

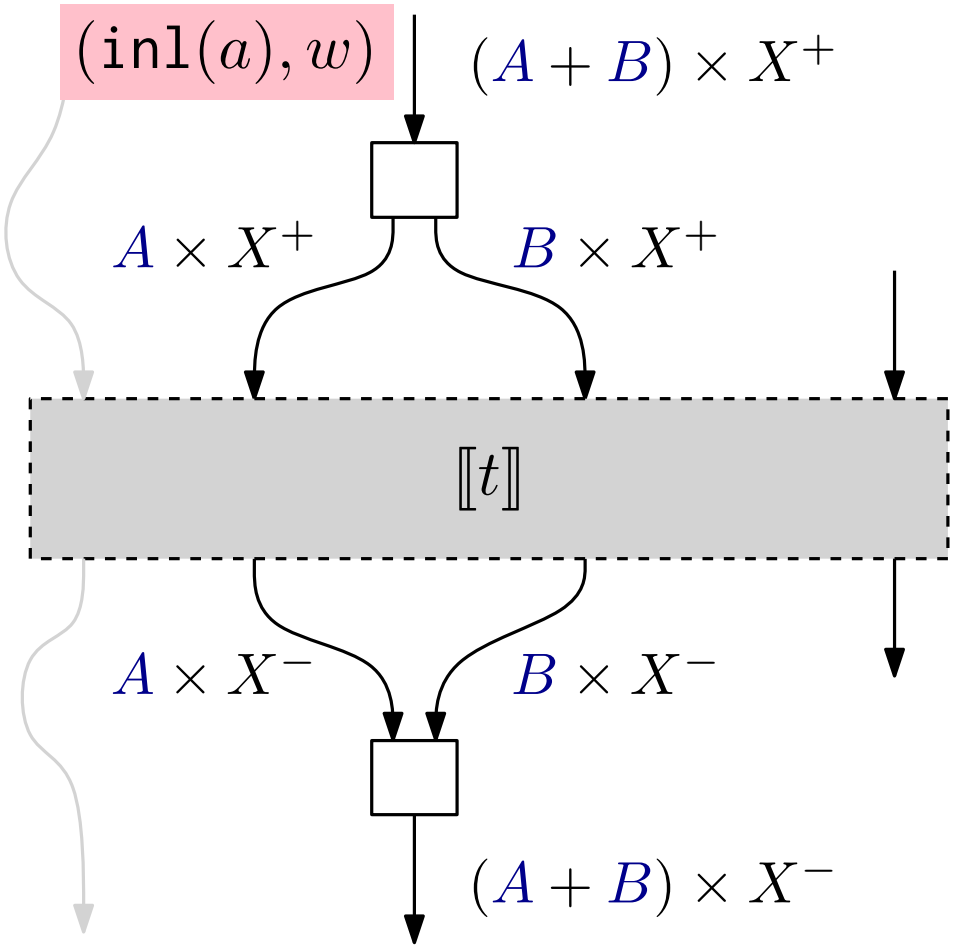
$[\![\mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma, x\colon A \cdot X, y\colon B \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, z\colon (A + B) \cdot X \vdash \mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathtt{in}\ t\colon Y}$$

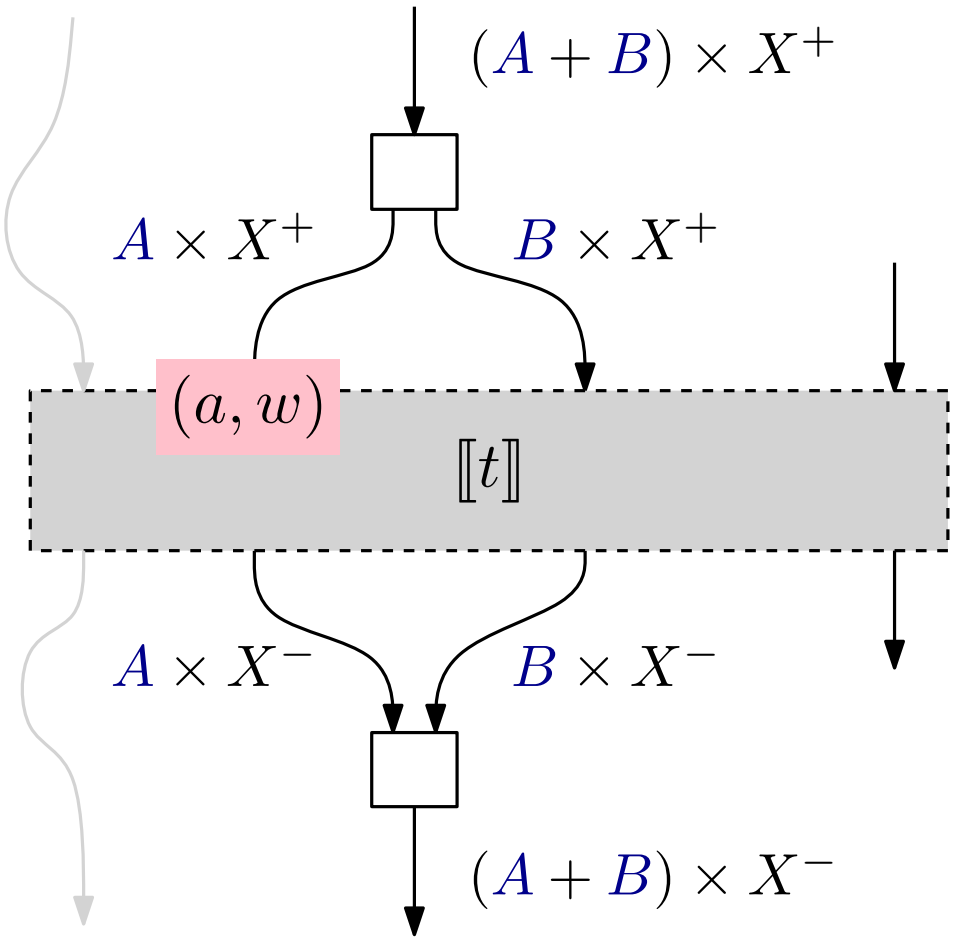
$[\![\mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathtt{in}\ t]\!] :=$



Translation

$$\frac{\Sigma \mid \Gamma, x\colon A \cdot X, y\colon B \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, z\colon (A + B) \cdot X \vdash \mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

$[\![\mathsf{copy}\ z\ \mathsf{as}\ x, y\ \mathsf{in}\ t]\!] :=$



Type Inference

Girard Translation:

$$X \to Y := (!X) \multimap Y$$

The typing rules of the simply-typed lambda calculus are derivable if one restricts to judgements of the form.

$$x_1 : !X_1, \ldots, x_n : !X_n \vdash t : X$$

Example:

$$\dfrac{\dfrac{!\Gamma \vdash s : !X \multimap Y \qquad \dfrac{!\Delta \vdash t : X}{!!\Delta \vdash t : !X}}{!\Gamma, !!\Delta \vdash s\, t : Y}}{!\Gamma, !\Delta \vdash s\, t : Y}$$

Type Inference

In INT we can be more precise.

Replace all occurrences of $!X$ by a $\alpha \cdot X$ for a fresh type variable α.

Example:

$$\dfrac{\dfrac{\dfrac{\alpha \cdot \Gamma \vdash s : \gamma \cdot X \multimap Y \qquad \dfrac{\beta \cdot \Delta \vdash t : X}{\gamma \cdot \beta \cdot \Delta \vdash t : \gamma \cdot X}}{\alpha \cdot \Gamma, \gamma \cdot \beta \cdot \Delta \vdash s\ t : Y}}{\alpha \cdot \Gamma, (\gamma \times \beta) \cdot \Delta \vdash s\ t : Y}}{\alpha \cdot \Gamma, \delta \cdot \Delta \vdash s\ t : Y}\ (\gamma \times \beta) \lhd \delta$$

Find an instantiation of the type variables that satisfies all $\lhd$-constraints.

Type Inference

Solving the $\lhd$-constraints (one possibility):

- All constraints have the form $A \lhd \alpha$.
- Gather all constraints with the same upper bound

$$A_1 \lhd \alpha, \quad \ldots, \quad A_n \lhd \alpha$$

- If α is not free in any A_i, then

$$\alpha := A_1 + \cdots + A_n$$

  is a solution. Otherwise

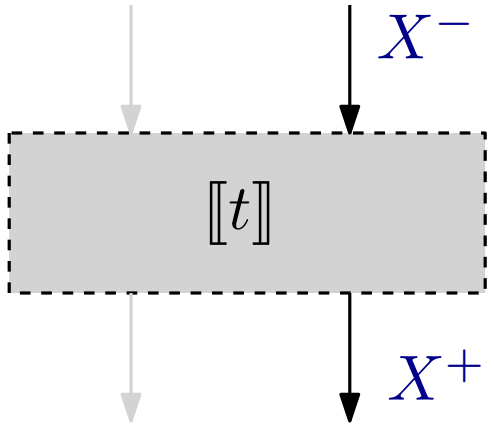
$$\alpha := \mu\alpha.\, A_1 + \cdots + A_n$$

  is a solution.
- Repeat until all constraints are solved.

Direct Definition

$$\frac{\Sigma \mid \Gamma \vdash t \colon X^- \to [X^+]}{\Sigma \mid \Gamma \vdash \mathtt{direct}_X(t) \colon X}$$

$[\![\mathtt{direct}_X(t)]\!] :=$

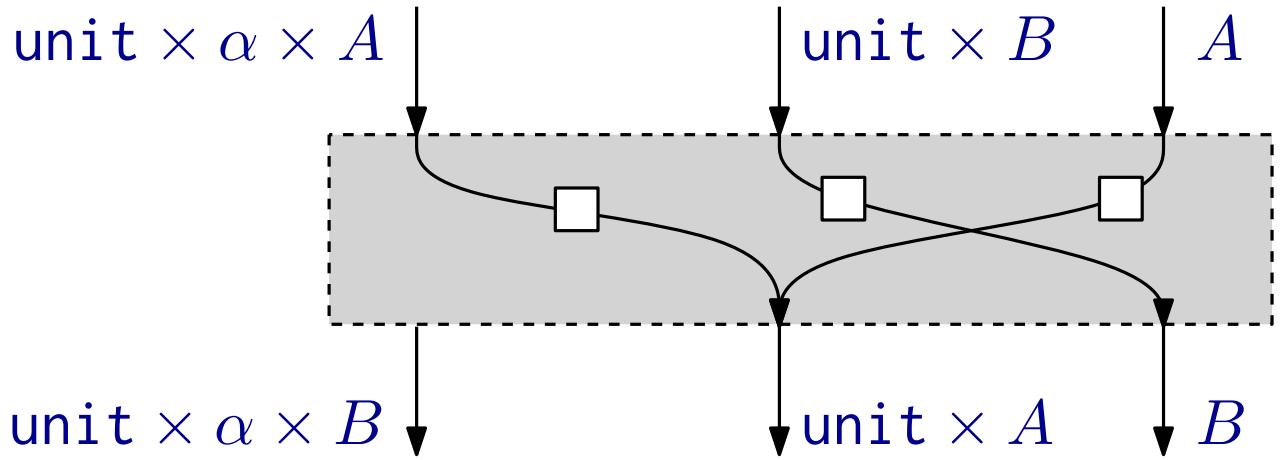


Examples:

- tail recursion
- recursion
- mutable variables
- coroutines
- callcc

Example — Tail Recursion

$$\texttt{tailrec}\colon \texttt{unit} \cdot (\alpha \cdot (A \to [B]) \multimap (A \to [B])) \multimap (A \to [B])$$



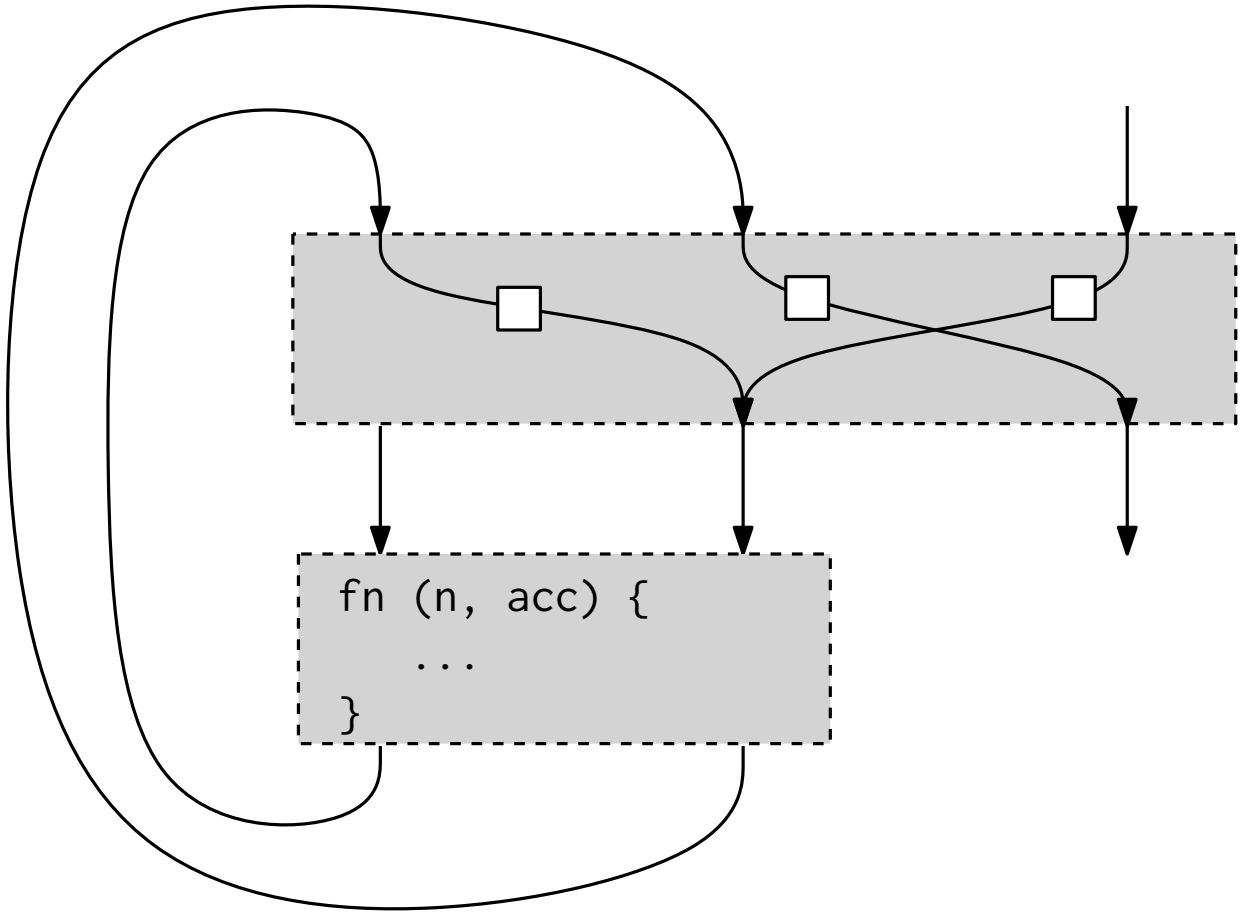

```
let tailrec = direct(
  case x of
      Inr(a) -> Inl((), Inr(a))
    | Inl(u, s) ->
        case s of
           Inr(b) -> Inr(b)
         | Inl(d, a) -> Inl((), Inr(a))
  )
```

Example — Tail Recursion

```
let facaux =
   tailrec (λ facaux ->
     fn (n, acc)  ->
       if n=0 then acc else facaux (n-1, n*acc)
   )

fn fac(n) =
   facaux (n, 1)
```

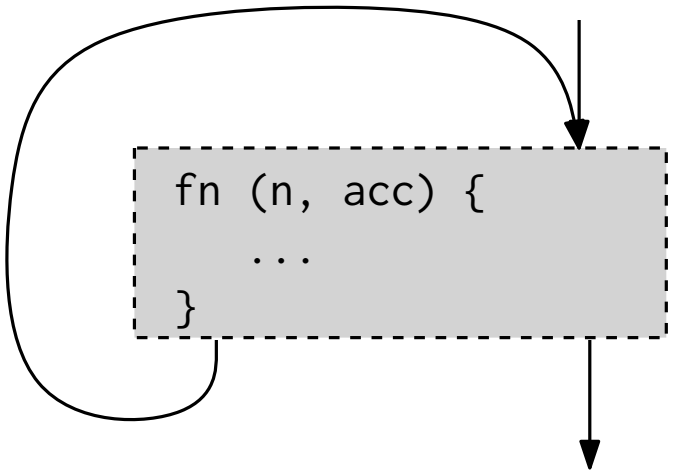
Example — Tail Recursion

```
let facaux =
    tailrec (λ facaux ->
      fn (n, acc)  ->
        if n=0 then acc else facaux (n-1, n*acc)
    )

fn fac(n) =
    facaux (n, 1)
```



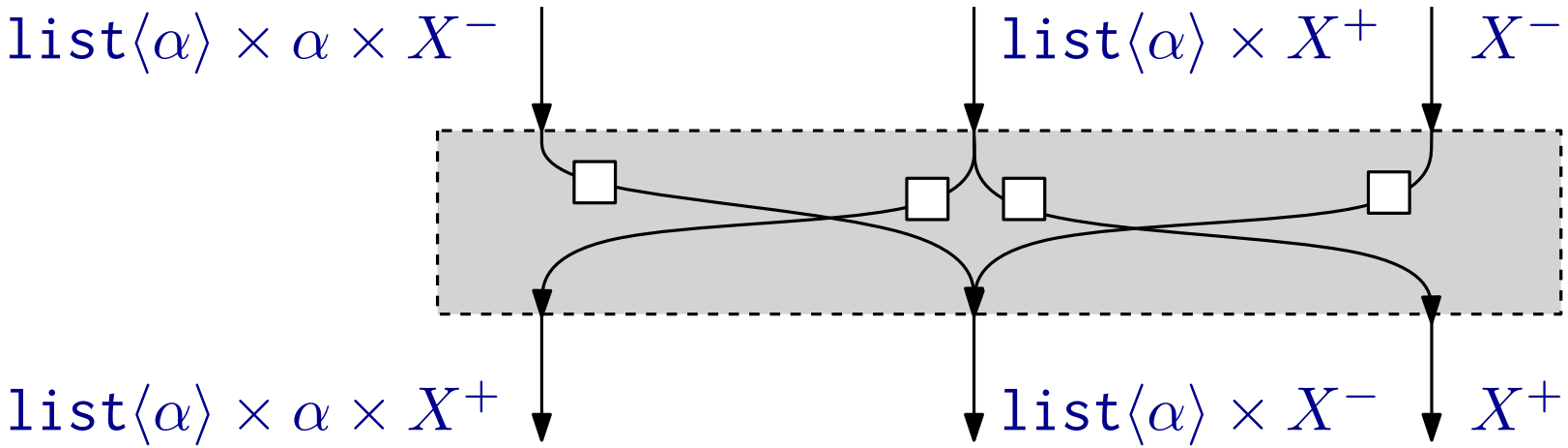
Example — Tail Recursion

```
let facaux =
    tailrec (λ facaux ->
      fn (n, acc)  ->
        if n=0 then acc else facaux (n-1, n*acc)
    )

fn fac(n) =
    facaux (n, 1)
```



Example — Recursion

$$\texttt{fix}\colon \texttt{list}\langle\alpha\rangle \cdot (\alpha \cdot X \multimap X) \multimap X$$

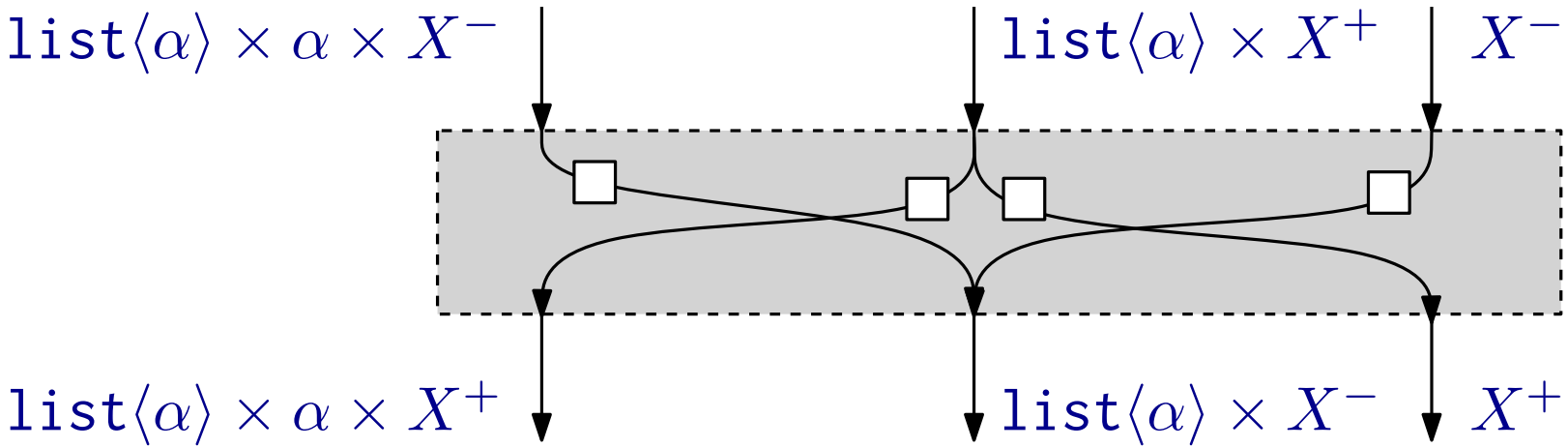


Example

```
let fib = fix (λ fib ->
   copy fib as fib1, fib2 in
   fn i ->
      if i < 2 then 1 else
        fib1(i-1) + fib2(i-2))
```

Example — Recursion

$$\texttt{fix}\colon \texttt{list}\langle\alpha\rangle \cdot (\alpha \cdot X \multimap X) \multimap X$$

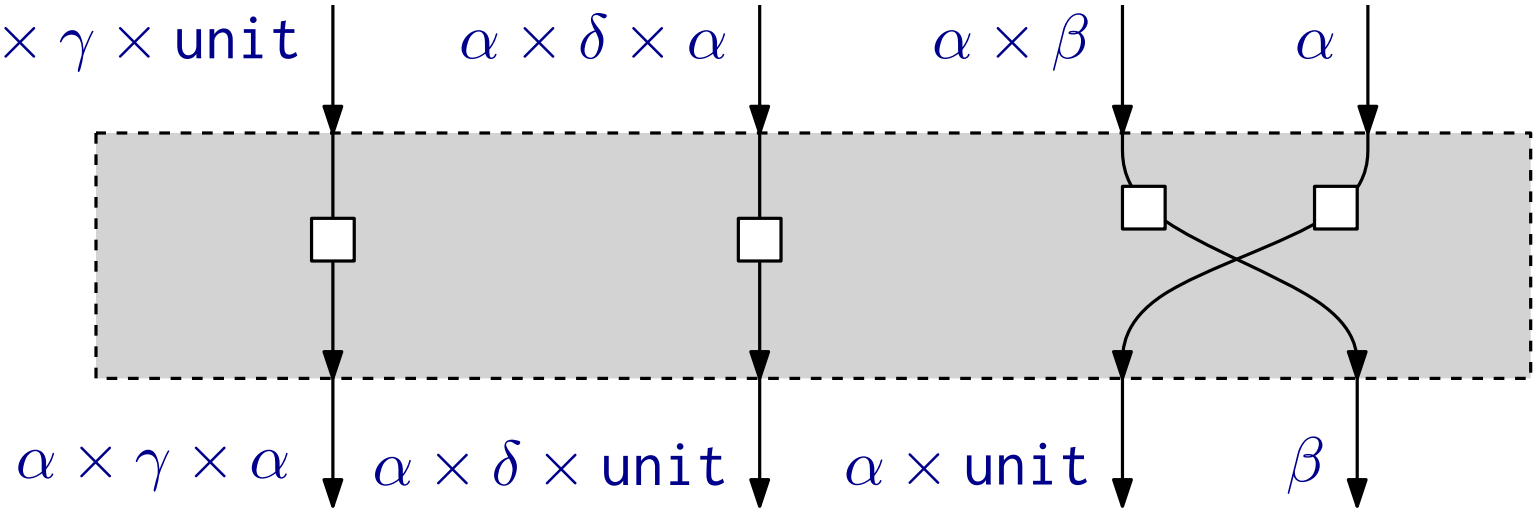


Example

```
let fib2 = fix (λ fib ->
  fn i -> tailrec (λ tr ->
            fn (i, acc)  ->
               if i < 2 then acc
               else let acc1 = fib(i-1) + acc in tr (i-2, acc1
            ) (i, 1))
```

Example — Stack Allocation

$$\texttt{newvar}\colon \alpha \cdot \left(\gamma \cdot [\alpha] \multimap \delta \cdot (\alpha \to [\texttt{unit}]) \multimap [\beta]\right) \multimap (\alpha \to [\beta])$$




```
newvar (λ read -> λ write ->
    write(5);
    let x = read in
      print x
) (6)
```

Example — Coroutines

$$\texttt{corout} : (\texttt{unit} + \delta_2) \cdot X_1 \multimap \delta_1 \cdot X_2 \multimap [\gamma]$$

where $X_1 = \delta_1 \cdot (\alpha \to [\beta]) \multimap [\gamma]$

$X_2 = \delta_2 \cdot (\beta \to [\alpha]) \multimap \alpha \to [\gamma]$

```
let main = corout proc1 proc2

let proc1 = λ yield ->
   tailrec (λ proc1 ->
      fn j ->
        print "p1:"; print j;
        let i = yield(j) in
        proc1(i)
    ) 3
```

```
let proc2 = λ yield ->
   copy yield as yield1, yield2 in
   tailrec (λ proc2 ->
     fn j ->
       print "p2:"; print j;
       let k = yield1(j+1) in
       let i = yield2(k+1) in
       proc2(i)
   )
```

Summary

INT is a higher-order calculus for composing low-level fragments.

- terms represent fragments
- types represent interfaces of fragments

Int construction / Geometry of Interaction

- few assumptions on low-level language
- expressive higher-order structure
- control over low-level details

Directions:

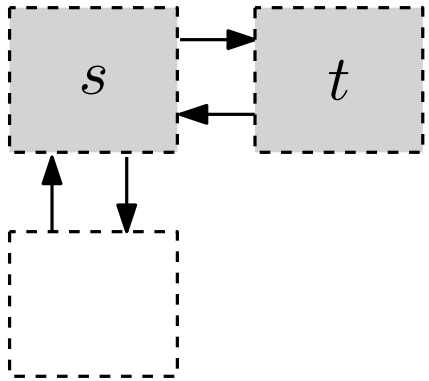
- correctness: $[\![t]\!] \sim_X (\!|f|\!)$
- decomposition, e.g. $[A]$ into $\forall\alpha.\,(A \to T\alpha) \multimap T\alpha$
- ...

Modules

A Simple Module System

INT is a calculus for composing low-level fragments like little modules.

$$\lambda x.\, s\ t$$



For writing programs, it is more convenient to reformulate these ideas by module system.

Int-construction / Geometry of Interaction:
A free module system for almost any low-level language.

Standard ML Modules

"[A] linking language for specifying how code fragments can be combined to form a complete program." [Elsman 1999]

Modules (Structures):

```
structure IntOrd =
    struct
      type t = int
      val compare = Int.compare
    end
```

Module Types (Signatures):

```
    sig
      type t = int
      val compare : t → t → int
    end
```

Standard ML Modules

```
signature ORD =
      sig
        type t
        val compare : t → t → int
      end
structure IntOrdSealed = IntOrd :> ORD

signature MAP =
      sig
        type key
        type α table

        val empty : α table
        val insert : key → α → α table → α table
        val lookup : key → α table → α option
      end
```

Standard ML Modules

Functors:

```
functor TreeMap(O : ORD) :> MAP where type key = O.t =
    struct
      open O
       ...
    end

structure IntMap = TreeMap(IntOrd)
```

F-ing Modules

Simple semantics of the ML module system by elaboration to System F_ω [Rossberg, Russo, Dreyer 2010].

Type declarations map to type variables, quantified appropriately.

- ```
  signature S =
    sig
      type t
      val f : int → t
    end
  ```

  $\exists\alpha.\,(\texttt{int} \to \alpha)$

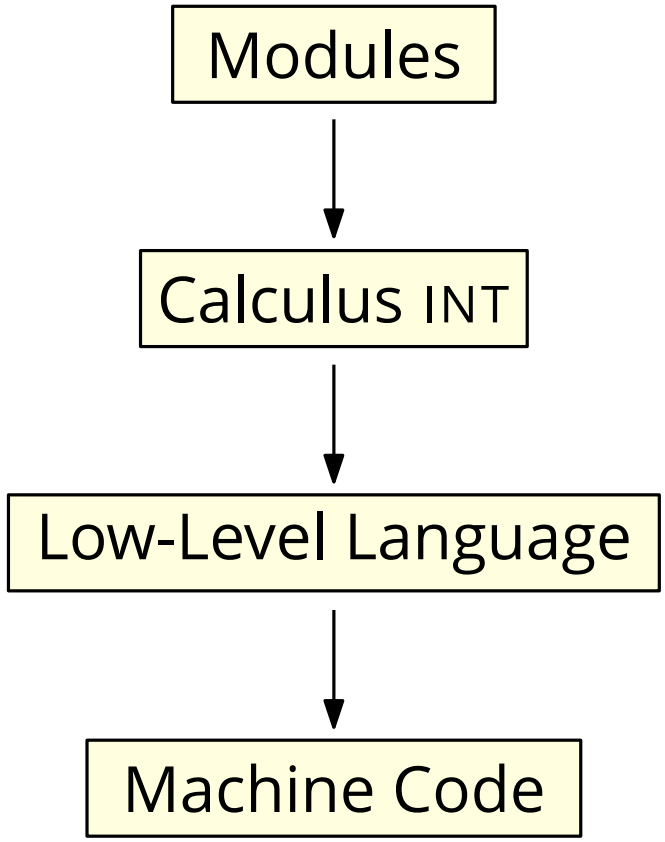
- ```
  functor (X: S) : S
  ```

  $\forall\alpha.\,(\texttt{int} \to \alpha) \to \exists\beta.\,(\texttt{int} \to \beta)$

A Simple Module System

The same idea can be applied to INT to define an ML-like module system for low-level programs.

Main advantage: ease of use for programming, compared to using explicit terms for ∀ and ∃.



A Simple Module System

Paths $p ::= X \mid p.X$

Value Types $A ::= p \mid$ low-level types

Signatures $S ::= A \to [B] \mid \texttt{type} \mid \texttt{type}\ A$

$\mid \texttt{sig}\ X{:}S; \ldots; X{:}S\ \texttt{end}$

$\mid \texttt{functor}(X{:}S) \to S$

Structures $m ::= \texttt{fn}\ x{:}A \to e \mid \texttt{type}\ A$

$\mid \texttt{struct}\ X = m; \ldots; X = m\ \texttt{end}$

$\mid \texttt{functor}(X : S) \to m \mid p \mid X\ X \mid X{:>}S$

Expressions $e ::= \texttt{return}\ v \mid \texttt{let}\ x = p(v)\ \texttt{in}\ e$

$\mid \texttt{let}\ x = \texttt{primop}(v)\ \texttt{in}\ e \mid \ldots$

$\mid \texttt{case}\ v\ \texttt{of}\ \texttt{inl}(y) \to e_1; \texttt{inr}(z) \to e_2$

Examples

Function definition

```
struct
    f = fn (x: int) → return (x, x)
end
```

is a module of type

```
sig
  f: int → [int × int]
end
```

Examples

Records

```
struct
    add = fn (x: int × int) →
              let (x1, x2) = x in
              let s = add(x1, x2) in
              return s;
    add2 = fn (z: int) → add(z, z)
end
```

is a module of type

```
sig
  add: int × int → [int];
  add2: int → [int]
end
```

Examples

Type declarations

```
struct
    t = type int;
    eval = (fn (x: int) → [x]) :> (int → [t]);
    apply = fn (z: t × int) →
                    let (x, y) = z in
                    let s = add(x, y) in
                    return s
end
```

is a module of type

```
sig
  t: type int;
  eval: int → [t];
  apply: t × int → [int]
end
```

Examples

Abstraction and Sealing

```
struct
    t = type int;
    eval = (fn (x: int) → [x]) :> (int → [t]);
    apply = fn (z: t × int) →
                    let (x, y) = z in
                    let s = add(x, y) in
                    return s
end :> (sig
            t: type;
            eval: int → [t];
            apply: t × int→ [int]
          end)
```

Examples

Functors and Paths

```
functor (X: sig t: type;
                eval: int → [t];
                apply: t × int → [int]
             end) →
  struct
     t = type (X.t × bool);
     eval = fn (x: int) → let y = X.eval(x) in
                          return (y, true)
  end
```

Examples

Higher-order Functors

Let `S` be `sig f : unit → [int] end`.

Then

```
functor (X: functor (Y: S) → S) →
struct
    Y1 = struct f = fn (x : unit) → return 3 end;
    Y2 = struct f = fn (x : unit) → return 5 end;
    X1 = X Y1;
    X2 = X Y2;
    f = fn (x: unit) →
          let x1 = X1.f() in let x2 = X2.f() in
          add(x1, x2)
end
```

has type `functor (X: functor (Y: S) → S) → S`.

Elaboration

Aim: Adapt the ideas from [Rossberg, Russo, Dreyer 2010] to translate the simple module system to INT.

$$\begin{array}{rcl} \text{signature} & \longmapsto & \text{INT type} \\ \text{structure} & \longmapsto & \text{INT term} \end{array}$$

Most of the module system maps to INT easily.

- $A \to [B]$ exists in both systems.
- `sig` $X_1{:}S_1; X_2{:}S_2$ `end` can be thought of as $S_1 \otimes S_2$.
- `functor`$(X{:}S_1)S_2$ can be thought of as $S_1 \multimap S_2$.

The difficulty is to account for `type`-declarations. These are all removed and replaced by appropriate quantification.

Elaboration

Example:

```
sig
  f: int × int → [int];
  g: int → [int]
end
```

corresponds to

$$(\texttt{int} \times \texttt{int} \to [\texttt{int}]) \otimes (\texttt{int} \to [\texttt{int}])$$

in INT.

Elaboration

Type declarations are removed.

Example:

```
sig
  t: type;
  T: sig
       t: type;
       f: int → [t]
     end;
  g: int × int → [T.t × t];
end
```

corresponds to

$$\exists\alpha.\,\exists\beta.\,(\texttt{int} \times [\beta]) \otimes (\texttt{int} \times \texttt{int} \to [\beta \times \alpha])$$

in INT.

Elaboration

To make such a translation precise, it is convenient to extend INT conservatively.

Record Types

$$\frac{\Sigma \mid \Gamma_1 \vdash t_1 : Y_1 \qquad \ldots \qquad \Sigma \mid \Gamma_n \vdash t_n : Y_n}{\Sigma \mid \Gamma_1, \ldots, \Gamma_n \vdash \langle f_1 = t_1; \ldots; f_n = t_n \rangle : \langle f_1 = X_1; \ldots; f_n = X_n \rangle}$$

$$\frac{\Sigma \mid \Gamma \vdash s : \langle f_1 = X_1; \ldots; f_n = X_n \rangle \qquad \Sigma \mid \Delta, x_1{:}X_1, \ldots, x_n{:}X_n \vdash t : Y}{\Sigma \mid \Gamma, \Delta \vdash \mathtt{let} \langle x_1, \ldots, x_n \rangle = s \ \mathtt{in}\ t : Y}$$

Labelled Singleton Types

$$\frac{}{\Sigma \mid \Gamma \vdash *_A : I_A}$$

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

```
sig
  t: type;
  T: sig
       t: type;
       f: int → [t]
     end;
  g: int × int → [T.t × t];
end
```

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

```
⟨
  t: type;
  T: sig
       t: type;
       f: int → [t]
     end;
  g: int × int → [T.t × t];
⟩
```

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

$\exists\alpha.\langle$
  t: I_α;
  T: sig
      t: type;
      f: int $\rightarrow$ [t]
    end;
  g: int $\times$ int $\rightarrow$ [T.t $\times$ t];
$\rangle$

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

$$
\begin{array}{l}
\exists\alpha.\langle \\
\quad \texttt{t}\colon I_\alpha; \\
\quad \texttt{T}\colon \langle \\
\qquad\quad \texttt{t}\colon \texttt{type}; \\
\qquad\quad \texttt{f}\colon \texttt{int} \to \texttt{[t]} \\
\qquad \rangle; \\
\quad \texttt{g}\colon \texttt{int} \times \texttt{int} \to \texttt{[T.t} \times \texttt{t]}; \\
\rangle
\end{array}
$$

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

$$
\begin{array}{l}
\exists\alpha.\,\exists\beta.\,\langle \\
\quad \texttt{t}\colon\ I_\alpha\,; \\
\quad \texttt{T}\colon\ \langle \\
\qquad\quad \texttt{t}\colon\ I_\beta\,; \\
\qquad\quad \texttt{f}\colon\ \texttt{int} \to \texttt{[t]} \\
\qquad \rangle\,; \\
\quad \texttt{g}\colon\ \texttt{int} \times \texttt{int} \to \texttt{[T.t} \times \texttt{t]}\,; \\
\rangle
\end{array}
$$

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

$$
\begin{array}{l}
\exists\alpha.\,\exists\beta.\,\langle \\
\quad \texttt{t}\colon\ I_\alpha\,; \\
\quad \texttt{T}\colon\ \langle \\
\qquad\quad \texttt{t}\colon\ I_\beta\,; \\
\qquad\quad \texttt{f}\colon\ \texttt{int} \to [\beta] \\
\qquad \rangle\,; \\
\quad \texttt{g}\colon\ \texttt{int} \times \texttt{int} \to [\texttt{T.t} \times \texttt{t}]\,; \\
\rangle
\end{array}
$$

Elaboration

Labelled singleton types are used to keep track of all the types that were removed.

Example:

$$
\begin{array}{l}
\exists\alpha.\,\exists\beta.\,\langle \\
\quad \mathtt{t}\colon\ I_\alpha\,; \\
\quad \mathtt{T}\colon\ \langle \\
\qquad\quad \mathtt{t}\colon\ I_\beta\,; \\
\qquad\quad \mathtt{f}\colon\ \mathtt{int} \to [\beta] \\
\qquad \rangle\,; \\
\quad \mathtt{g}\colon\ \mathtt{int} \times \mathtt{int} \to [\beta \times \alpha]\,; \\
\rangle
\end{array}
$$

Examples

Example

```
functor (X: sig
              t: type;
              eval: int → t;
              apply: t × int → [int]
            end) →
struct
   t = type (X.t × bool);
   eval = fn (x: int) : t → ...
end
```

Examples

Example

```
functor (X: ∃α.⟨
              t: I_α;
              eval: int → α;
              apply: α × int → [int]
            ⟩) →
⟨
    t = type (X.t × bool);
    eval = fn (x: int) : t → ...
⟩
```

Examples

Example

```
∀α. functor (X: ⟨
                  t: I_α;
                  eval: int → α;
                  apply: α × int → [int]
                ⟩) →
   ⟨
      t = type (X.t × bool);
      eval = fn (x: int) : X.t × bool → ...
   ⟩
```

Examples

Example

```
∀α. functor (X: ⟨
                  t: I_α;
                  eval: int → α;
                  apply: α × int → [int]
                ⟩) →
  ⟨
     t = *: I_{α × bool};
     eval = fn (x: int) : α × bool → ...
  ⟩
```

Elaboration

Translation to INT (with record types).

$$\Xi ::= \exists \vec{\alpha}. \Sigma$$
$$\Sigma ::= I_A \mid !(A \to [B]) \mid !\langle f{:}\Sigma, \ldots, f{:}\Sigma \rangle \mid !(\forall \vec{\alpha}. \Sigma \multimap \Xi)$$

For each syntactic category, there is an elaboration judgement:

- Signatures: $\Gamma \vdash S \rightsquigarrow \Xi$
- ...

Adaptation of “F-ing Modules” [Rossberg, Russo, Dreyer 2010].

Elaboration

Signatures

$$\frac{\Gamma \vdash A \rightsquigarrow A' \qquad \Gamma \vdash B \rightsquigarrow B'}{\Gamma \vdash A \to [B] \rightsquigarrow !(A' \to [B'])}$$

$$\frac{}{\Gamma \vdash \mathsf{type} \rightsquigarrow \exists \alpha.\, I_\alpha}$$

$$\frac{\Gamma \vdash A \rightsquigarrow A'}{\Gamma \vdash \mathsf{type}\ A \rightsquigarrow I_{A'}}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists \vec{\alpha_1}.\, \Sigma_1 \qquad \ldots \qquad \Gamma \vdash S_n \rightsquigarrow \exists \vec{\alpha_n}.\, \Sigma_n}{\Gamma \vdash \mathsf{sig}\ \overrightarrow{X_i : S_i}\ \mathsf{end} \ \rightsquigarrow\ \exists \vec{\alpha_1}, \ldots \vec{\alpha_n}.\, !\langle \overrightarrow{X_i : \Sigma_i} \rangle}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists \vec{\alpha_1}.\, \Sigma_1 \qquad \Gamma,\, X{:}\Sigma_1 \vdash S_2 \rightsquigarrow \exists \vec{\alpha_2}.\, \Sigma_2}{\Gamma \vdash \mathsf{functor}(X : S_1) \to S_2 \ \rightsquigarrow\ !(\forall \vec{\alpha}_1.\, \Sigma_1 \multimap \exists \alpha_2.\, \Sigma_2)}$$

Elaboration

Types (selection)

$$\frac{\Gamma(X) = I_A}{\Gamma \vdash X \rightsquigarrow A}$$

$$\frac{\Gamma \vdash A \rightsquigarrow A' \qquad \Gamma \vdash B \rightsquigarrow B'}{\Gamma \vdash A \times B \rightsquigarrow A' \times B'}$$

Elaboration

Structures (selection)

$$\frac{\Gamma \vdash A\ \mathsf{type} \rightsquigarrow A'}{\Gamma \vdash \mathsf{type}\ A \colon I_{A'} \rightsquigarrow *}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists \vec{\alpha}.\, \Sigma \qquad \Gamma,\, \vec{\alpha},\, X{:}\Sigma \vdash m \colon \Xi \rightsquigarrow t}{\Gamma \vdash \mathsf{functor}(X : S) \to m \colon \forall \vec{\alpha}.\, !(\Sigma \multimap \Xi) \rightsquigarrow \Lambda \vec{\alpha}.\, \lambda X.\, t}$$

$$\frac{\Gamma(X) = !(\forall \vec{\alpha}.\Sigma \multimap \Xi) \qquad \Gamma(Y) = \Sigma' \qquad \Gamma \vdash \Sigma' \leq \exists \vec{\alpha}.\Sigma \uparrow \vec{\tau} \rightsquigarrow f}{\Gamma \vdash X\ Y \colon \Xi[\vec{\tau}/\vec{\alpha}] \rightsquigarrow X\ \vec{\tau}\ (f\ Y)}$$

$$\frac{\Gamma, Z_1{:}\Sigma_1, \ldots, Z_n{:}\Sigma_n, \Delta \vdash m \colon \Xi \rightsquigarrow t}{\Gamma, X{:}!\langle Y_1{:}\Sigma_1; \ldots; Y_n{:}\Sigma_n\rangle,\, \Delta \vdash m[\overrightarrow{X.Y/Z}] \colon \Xi \rightsquigarrow \mathsf{let}\ \langle \vec{Z} \rangle = X\ \mathsf{in}\ t}$$

Fine Points

Where do we insert copy-operations?

- Example:

```
functor (X: sig f: ...; g: ... end) →
  sig Y1 = X.f; Y2 = X.f; Y3 = X.g end
```

  Do we duplicate `X` or just `X.f`?
- Can make a default choice or allow manual duplication.

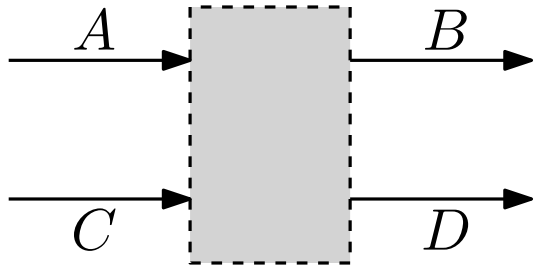
More precise low-level implementation:

- subexponentials
- bounded quantification
- type inference

Summary

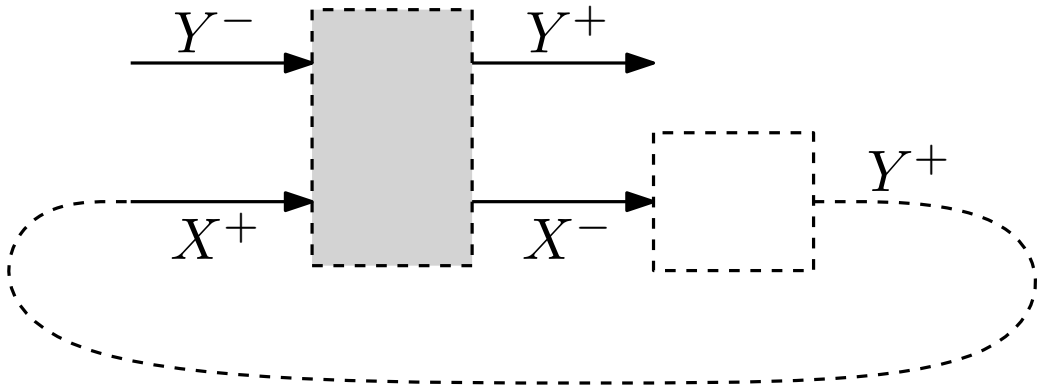
A free module system for almost any low-level language!

```
sig
  f: A → [B];
  g: C → [D]
end
```



Linking of program fragments is functor application.

```
functor (X: X) → Y
```



Duplication of modules handled by subexponentials (tagging).

Compilation

PCF

Simply-typed λ-calculus with base types and constants:

$$
\begin{aligned}
X, Y &::= 1 \mid \mathbb{N} \mid X \to Y \\
s, t &::= x \mid \lambda x{:}X.\, t \mid s\ t \\
&\mid n \mid s + t \mid \texttt{if0}\ s\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \\
&\mid \texttt{let}\ x = s\ \texttt{in}\ t \mid \texttt{fix}(t)
\end{aligned}
$$

Consider different variants:

- call-by-name
- call-by-value

Call-by-Name

The obvious translation to INT correctly implements call-by-name.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x{:}X.\, t]\!] &= \lambda x.\, \mathsf{copy}\ x\ \mathsf{as}\ \vec{x}\ \mathtt{in}\ [\![t_{\vec{x},x}]\!] \\
[\![s\ t]\!] &= [\![s]\!]\ [\![t]\!] \\
[\![\mathtt{fix}(t)]\!] &= \mathtt{fix}([\![t]\!]) \\
[\![n]\!] &= \mathtt{return}\ n \\
[\![s + t]\!] &= \mathsf{add}\ [\![s]\!]\ [\![t]\!] \\
[\![\mathtt{let}\ x = s\ \mathtt{in}\ t]\!] &= [\![(\lambda x.\, t)\ s]\!]
\end{aligned}
$$

Types:

$$
\begin{aligned}
[\![\mathbb{N}]\!] &= [\mathtt{int}] \\
[\![X \to Y]\!] &= A \cdot [\![X]\!] \multimap [\![Y]\!]
\end{aligned}
$$

Related to CPS-translation and defunctionalisation [S. 2014].

Call-by-Value

The translation of call-by-value is less immediate

For terms of function type, we must distinguish between *evaluation* and *application*.

Examples:

```
(fun f → f 1 + f 2) (let y = expensive() in fun x → x + y)

(fun f → f 1 + f 2) (print "hello"; fun x → x + 1)
```

Call-by-Value

What doesn't work (well):

- "call-by-value" Girard translation to INT
  (evaluates terms more than once)
- CPS-translation to call-by-name
  (correct; space usage problematic)
- naive implementation of call-by-value games
  (memory management?)

Call-by-Value — What do Compilers implement?

Example

```
let a = 2 in
let b = a + 3 in
let f = fun x → a * x + b in
let g = if a < b then f else fun x → x + 3 in
print (g 5)
```

Call-by-Value — What do Compilers implement?

Example

```
let a = 2 in
let b = a + 3 in
let f = ... in
let g = if a < b then f else ... in
print (apply(g, 5))

fun apply(g, x) = ...

fun apply1(..., x) = a * x + b

fun apply2(..., x) = x + 3
```

Call-by-Value — What do Compilers implement?

Closures

```
let a = 2 in
let b = a + 3 in
let f = {addr = &apply1; vars = (a, b)} in
let g = if a < b then f else {addr = &apply2; vars = ()} in
print (apply(g, 5))

fun apply(c, x) = (*c.addr)(c.vars, x)

fun apply1((a, b), x) = a * x + b

fun apply2((), x) = x + 3
```

Call-by-Value — What do Compilers implement?

Defunctionalisation

```
let a = 2 in
let b = a + 3 in
let f = (a, b) in
let g = if a < b then Left(f) else Right() in
print (apply(g, 5))

fun apply(g, x) =
   case g of
   | Left(f) → apply1(f, x)
   | Right() → apply2((), x)

fun apply1((a, b), x) = a * x + b

fun apply2((), x) = x + 3
```

Call-by-Value

Idea: Translate a closed PCF term $t : X$ to a module of signature

$\mathcal{M}[\![X]\!]$:= sig
  T: $\mathcal{I}[\![X]\!]$,
  eval: unit $\to$ T.t
end

Call-by-Value

Idea: Translate a closed PCF term $t : X$ to a module of signature

$\mathcal{M}[\![X]\!]$:= sig
  T: $\mathcal{I}[\![X]\!]$,
  eval: unit $\to$ T.t
end

Base Type

$\mathcal{I}[\![\mathbb{N}]\!]$ = sig
  t: type int
end

Call-by-Value

Idea: Translate a closed PCF term $t : X$ to a module of signature

$\mathcal{M}[\![X]\!]$:= sig
  T: $\mathcal{I}[\![X]\!]$,
  eval: unit $\to$ T.t
end

Function types (special case)

$\mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]$ = sig
  t: type (* abstract *)
  apply: t $\times$ int $\to$ int
end

Call-by-Value

Idea: Translate a closed PCF term $t : X$ to a module of signature

$\mathcal{M}[\![X]\!]$:= sig
  T: $\mathcal{I}[\![X]\!]$,
  eval: unit $\to$ T.t
end

Function type (special case)

$\mathcal{I}[\![(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}]\!]$ = sig
  t: type (* abstract *)
  T: functor (X: $\mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]$) $\to$ sig
    apply: t $\times$ X.t $\to$ int
  end
end

Call-by-Value

Idea: Translate a closed PCF term $t\colon X$ to a module of signature

$\mathcal{M}[\![X]\!]$:= sig
  T: $\mathcal{I}[\![X]\!]$,
  eval: unit $\to$ T.t
end

Function type (special case)

$\mathcal{I}[\![(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})]\!]$ = sig
  t: type (* abstract *)
  T: functor (X: $\mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]$) $\to$ sig
    T : $\mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]$
    apply: t $\times$ X.t $\to$ T.t
  end
end

Call-by-Value

Idea: Translate a closed PCF term $t\colon X$ to a module of signature

$\mathcal{M}[\![X]\!]$:= sig
  T: $\mathcal{I}[\![X]\!]$,
  eval: unit $\to$ T.t
end

Function types (general case)

$\mathcal{I}[\![X \to Y]\!]$ = sig
  t: type (* abstract *)
  T: functor (X: $\mathcal{I}[\![X]\!]$) $\to$ sig
    T : $\mathcal{I}[\![Y]\!]$
    apply: t $\times$ X.t $\to$ T.t
  end
end

Call-by-Value

A PCF term $x_1{:}X_1, \ldots, x_k{:}X_k \vdash t : Y$ translates to a module of type

```
sig
  T:  𝓘⟦Y⟧,
  eval: X₁.t × ··· × Xₙ.t → T.t
end
```

that may make use of module variables $X_1 : \mathcal{I}[\![X_1]\!]$, …, $X_n : \mathcal{I}[\![X_n]\!]$.

Call-by-Value

A PCF term $x_1{:}X_1, \ldots, x_k{:}X_k \vdash t : Y$ translates to a module of type

```
sig
  T:  ⟦Y⟧,
  eval: X1.t × ··· × Xn.t → T.t
end
```

that may make use of module variables $X_1 : \mathcal{I}[\![X_1]\!], \ldots, X_n : \mathcal{I}[\![X_n]\!]$.

Recall:

```
I⟦ℕ⟧ = sig
    t: type int
end
```

```
I⟦X → Y⟧ = sig
  t: type (* abstract *)
  T: functor (X: I⟦X⟧) → sig
       T : I⟦Y⟧
       apply: t × X.t → T.t
     end
end
```

Call-by-Value

A PCF term $x_1{:}X_1, \ldots, x_k{:}X_k \vdash t\colon Y$ translates to a module of type

```
sig
  T:  𝓘⟦Y⟧,
  eval: X₁.t × ⋯ × Xₙ.t → T.t
end
```

that may make use of module variables $X_1\colon \mathcal{I}[\![X_1]\!], \ldots, X_n\colon \mathcal{I}[\![X_n]\!]$.

There is more than one way of defining such a translation, e.g.

- using heap-allocated closures (needs to add pointers to the low-level language)
- defunctionalisation

Defuctionalisation — Application

$$\text{APP}\ \frac{\Gamma \vdash s : X \to Y \qquad \Delta \vdash t : X}{\Gamma, \Delta \vdash s\ t : Y}$$

```
struct
  Ts = ... (* translation of s *)
  Tt = ... (* translation of t *)
  Y = Ts.T(Tt);

  T = Y.T;
  eval = fn (x⃗, y⃗) →
    let f = Ts.eval(x⃗) in
    let x = Tt.eval(y⃗) in
    Y.apply(f, x)
end
```

Recall:

```
𝓘⟦X → Y⟧ = sig
  t: type; (* abstract *)
  T: functor (X: 𝓘⟦X⟧) → sig
       T : 𝓘⟦Y⟧;
       apply: t × X.t → T.t
     end
end
```

Defuctionalisation— Abstraction

$$\textsc{abs}\ \frac{\Gamma,\, x{:}X \vdash t{:}\, Y}{\Gamma \vdash \mathsf{fn}\, x \Rightarrow t{:}\, X \to Y}$$

```
struct
  T = functor (X: 𝓘⟦X⟧) → struct
    T = ... (* translation of t *)
    apply = fn (f, x) →
      let (γ⃗) = f in
      T.eval(γ⃗, x)
  end;

  eval = fn (γ⃗) → return (γ⃗)
end
```

Recall:

```
𝓘⟦X → Y⟧ = sig
  t: type; (* abstract *)
  T: functor (X: 𝓘⟦X⟧) → sig
       T : 𝓘⟦Y⟧;
       apply: t × X.t → T.t
     end
end
```

Defunctionalisation — If

$$\textsf{IF}\ \frac{\vdash t : \mathbb{B} \qquad \vdash s_1 : \mathbb{N} \to \mathbb{N} \qquad \vdash s_2 : \mathbb{N} \to \mathbb{N}}{\vdash \texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \mathbb{N} \to \mathbb{N}}$$

```
struct
  Tt = ... (* translation of t *)
  Ts1 = ... (* translation of s *)
  Ts2 = ... (* translation of s *)

  type t = Left of Ts1.t | Right of Ts2.t
  eval = fn () →
    if Tt.eval() then Left(Ts1.eval())
    else Right(Ts2.eval());
  apply = fn(f, x) → case f of
        Left(f1) → Ts1.apply(f1, x)
      | Right(f2) → Ts2.apply(f2, x)
end
```

Recall:

```
ℐ⟦ℕ → ℕ⟧ = sig
  t: type (* abstract *)
  apply: t × int → int
end
```

(simplified)

Low-Level Representation of Values

signature	INT-elaboration
$\mathcal{I}[\![\mathbb{N}]\!]$ = sig t: type int end	$\langle\rangle$

$\mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]$ = sig
  t: type (* abstract *)
  apply: t $\times$ int $\to$ int
end
	$\exists\alpha.!\langle \mathsf{apply} : !(\alpha \times \mathsf{int} \to [\mathsf{int}])\rangle$

$\mathcal{I}[\![(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}]\!]$ = sig
  t: type (* abstract *)
  T: functor (X: $\mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]$) $\to$ sig
  apply: t $\times$ X.t $\to$ int
  end
end
	$\exists\alpha.!\langle \mathsf{T} : \forall\beta.$
	$!\langle \mathsf{apply} : !(\beta \times \mathsf{int} \to \mathsf{int})\rangle$
	$\multimap$
	$!\langle \mathsf{apply} : !(\alpha \times \beta \to \mathsf{int})\rangle$

Low-Level Representation of Values

	INT-elaboration
	$\langle\rangle$
	$\exists\alpha.\,!\langle \texttt{apply} : !(\alpha \times \texttt{int} \to [\texttt{int}])\rangle$
	$\exists\alpha.\,!\langle \texttt{T} : \forall\beta.\; !\langle \texttt{apply} : !(\beta \times \texttt{int} \to \texttt{int})\rangle \multimap\; !\langle \texttt{apply} : !(\alpha \times \beta \to \texttt{int})\rangle$

Low-Level Representation of Values

INT-elaboration	
$\langle\rangle$	
$\exists\alpha.!\langle\mathsf{apply} : !(\alpha \times \mathtt{int} \to [\mathtt{int}])\rangle$	
$\exists\alpha.!\langle\mathsf{T} : \forall\beta.\ !\langle\mathsf{apply} : !(\beta \times \mathtt{int} \to \mathtt{int})\rangle \multimap\ !\langle\mathsf{apply} : !(\alpha \times \beta \to \mathtt{int})\rangle$	

Low-Level Representation of Values

INT-elaboration	annotation inference
$\langle\rangle$	$\langle\rangle$
$\exists\alpha.\,!\langle \mathsf{apply} : !(\alpha \times \mathtt{int} \to [\mathtt{int}])\rangle$	$\exists\alpha \lhd A.$ $\quad B \cdot \langle \mathsf{apply} : C \cdot (\alpha \times \mathtt{int} \to [\mathtt{int}])$
$\exists\alpha.\,!\langle \mathsf{T} : \forall\beta.$ $\quad !\langle \mathsf{apply} : !(\beta \times \mathtt{int} \to \mathtt{int})\rangle$ $\quad \multimap$ $\quad !\langle \mathsf{apply} : !(\alpha \times \beta \to \mathtt{int})\rangle$	$\exists\alpha \lhd A.$ $C \cdot \langle \mathsf{T} : \forall\beta \lhd B.$ $\quad D \cdot \langle \mathsf{apply} : E \cdot (\beta \times \mathtt{int} \to \mathtt{int})\rangle$ $\quad \multimap$ $\quad F \cdot \langle \mathsf{apply} : G \cdot (\alpha \times \beta \to \mathtt{int})\rangle$

Low-Level Representation of Values

	annotation inference
	$\langle\rangle$
	$\exists\alpha \lhd A.$ $B \cdot \langle \mathtt{apply} : C \cdot (\alpha \times \mathtt{int} \to [\mathtt{int}])$
	$\exists\alpha \lhd A.$ $C \cdot \langle \mathsf{T} : \forall\beta \lhd B.$ $D \cdot \langle \mathtt{apply} : E \cdot (\beta \times \mathtt{int} \to \mathtt{int}) \rangle$ $\multimap$ $F \cdot \langle \mathtt{apply} : G \cdot (\alpha \times \beta \to \mathtt{int}) \rangle$

Low-Level Representation of Values

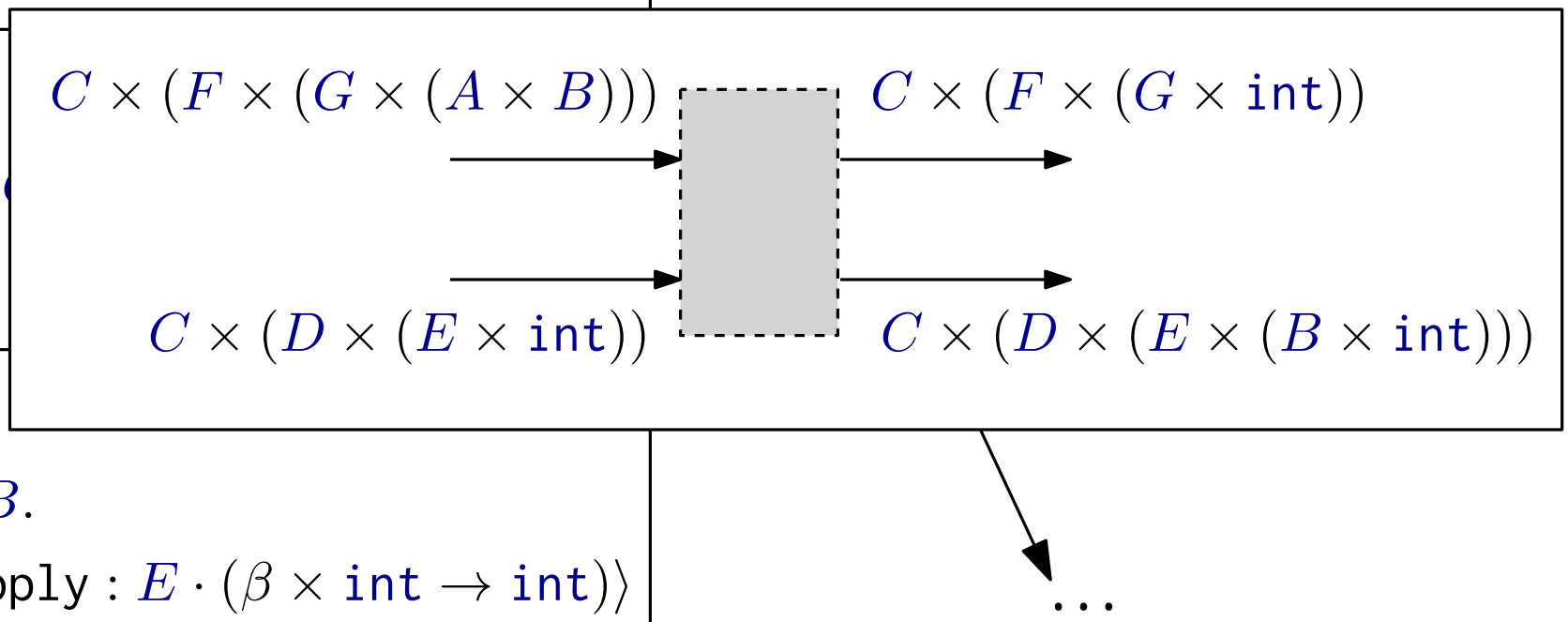
annotation inference	
$\langle\rangle$	
$\exists\alpha \lhd A.$ $\quad B \cdot \langle \mathsf{apply} : C \cdot (\alpha \times \mathtt{int} \to [\mathtt{int}])$	
$\exists\alpha \lhd A.$ $C \cdot \langle \mathsf{T} : \forall\beta \lhd B.$ $\qquad D \cdot \langle \mathsf{apply} : E \cdot (\beta \times \mathtt{int} \to \mathtt{int}) \rangle$ $\qquad \multimap$ $\qquad F \cdot \langle \mathsf{apply} : G \cdot (\alpha \times \beta \to \mathtt{int}) \rangle$	

Low-Level Representation of Values

annotation inference	program fragment
$\langle\rangle$	
$\exists\alpha \lhd A.$ $B \cdot \langle \mathsf{apply} : C \cdot (\alpha \times \mathtt{int} \to [\mathtt{int}])$	$B \times (C \times (A \times \mathtt{int}))$  $B \times C \times \mathtt{int}$
$\exists\alpha \lhd A.$ $C \cdot \langle \mathsf{T} : \forall\beta \lhd B.$ $D \cdot \langle \mathsf{apply} : E \cdot (\beta \times \mathtt{int} \to \mathtt{int}) \rangle$ $\multimap$ $F \cdot \langle \mathsf{apply} : G \cdot (\alpha \times \beta \to \mathtt{int}) \rangle$	...

Low-Level Representation of Values

annotation inference	program fragment
$\langle\rangle$	
$\exists\alpha \lhd A.$ $B \cdot \langle \texttt{apply} : ($	
$\exists\alpha \lhd A.$ $C \cdot \langle \texttt{T} : \forall\beta \lhd B.$ $D \cdot \langle \texttt{apply} : E \cdot (\beta \times \texttt{int} \to \texttt{int})\rangle$ $\multimap$ $F \cdot \langle \texttt{apply} : G \cdot (\alpha \times \beta \to \texttt{int})\rangle$	



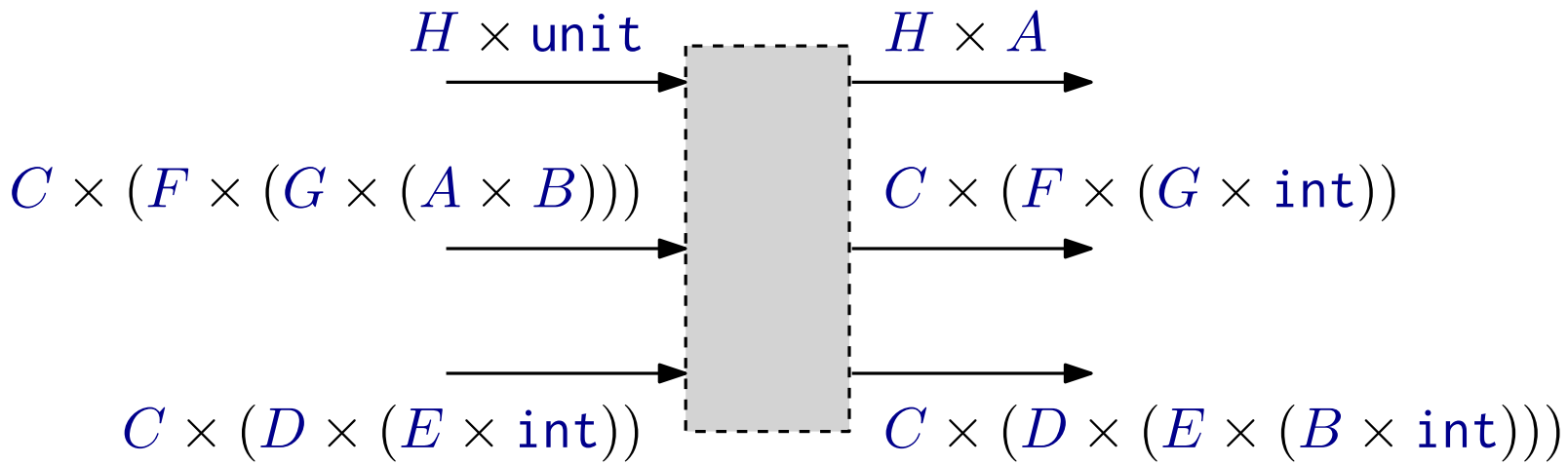
Low-Level Representation of Terms

Example

A PCF term $t\colon (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ translates to a module of signature

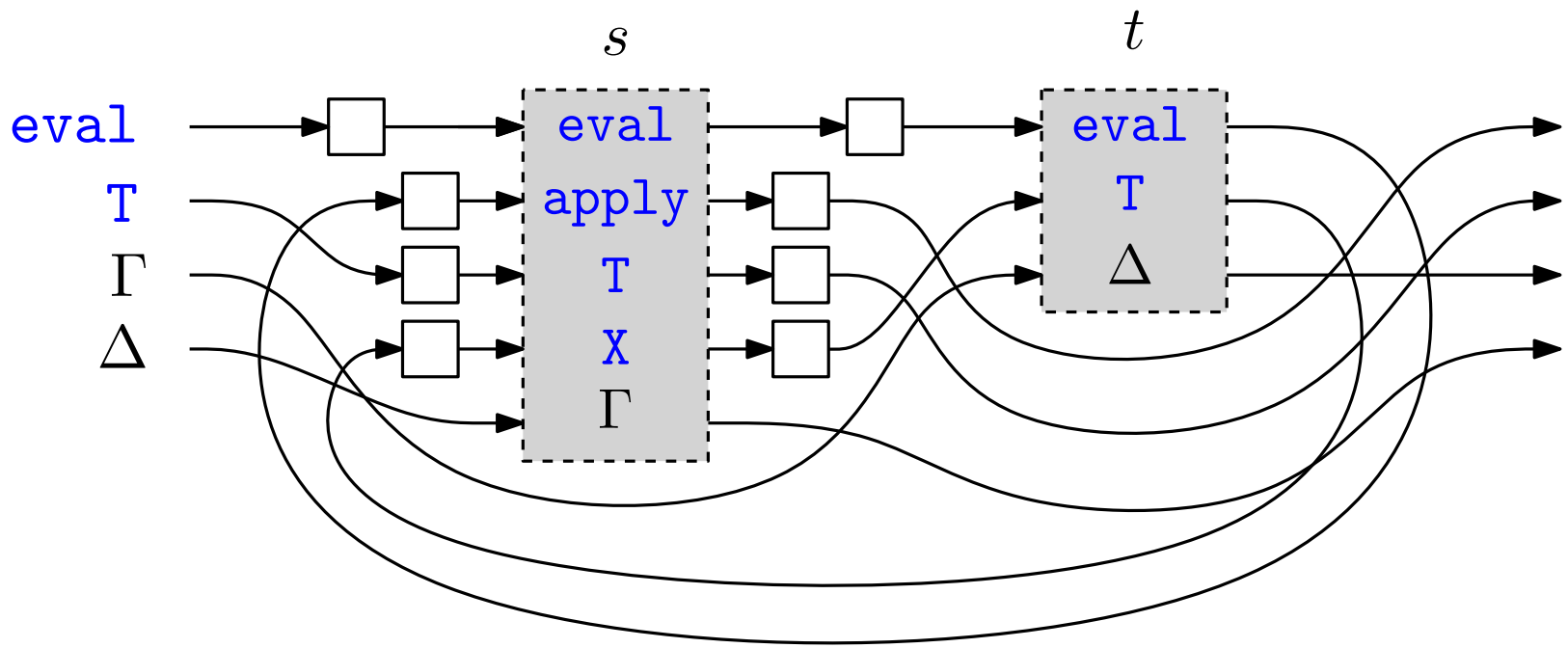
```
sig
  T: 𝓘⟦(ℕ → ℕ) → ℕ⟧;
  eval: unit → T.t
end
```

The low-level interface is $\mathcal{I}[\![(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}]\!]$ with an additional entry- and exit-point for `eval`.



Low-Level Representation — Application

$$\textsc{app}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$



Low-Level Representation — Abstraction

$$\textsf{ABS}\ \frac{\Gamma,\, x{:}X \vdash t{:}\, Y}{\Gamma \vdash \textsf{fn}\, x \Rightarrow t{:}\, X \to Y}$$