DEFUNCTIONALISATION AS MODULAR CLOSURE CONVERSION

Ulrich Schöpp
LMU Munich

MOTIVATION

Systematic understanding of the low-level aspects of real-world programming language implementations.

- formal compiler verification
- resource usage analysis and certification
- modularity and scalability
- ...

Today: Implementation of higher-order functions by first-order ones

HEAP-ALLOCATED CLOSURES

Heap-allocated closures

- Encode functions as pairs `(addr, env)`.
- Function values become pointers to such pairs.

Example

```
...
let f = fun x → a * x + b in
let g = if a < b then f else fun x → x + 3 in
print ((f 4) + (g 5))
```

HEAP-ALLOCATED CLOSURES

Heap-allocated closures

- Encode functions as pairs `(addr, env)`.
- Function values become pointers to such pairs.

Example

```
...
let f = fun x → a * x + b in
let g = if a < b then f else fun x → x + 3 in
print ((f 4) + (g 5))

fun apply1((a, b), x) = a * x + b
fun apply2((), x) = x + 3
```

HEAP-ALLOCATED CLOSURES

Heap-allocated closures

- Encode functions as pairs `(addr, env)`.
- Function values become pointers to such pairs.

Example

```
...
let f = (&apply1, (a, b)) in
let g = if a < b then f else (&apply2, ()) in
print (apply(f, 4) + apply(g, 5))

fun apply1((a, b), x) = a * x + b
fun apply2((), x) = x + 3

fun apply((addr, env), x) = (*addr)(env, x)
```

Defunctionalisation

- Replace the address by a tag that identifies the function.
- Use an algebraic data type to exclude invalid pairs `(tag, env)`: The pair becomes a constructor application `Ctag(env)`.

Example

```
...
let f = fun x → a * x + b in
let g = if a < b then f else fun x → x + 3 in
print ((f 4) + (g 5))
```

Defunctionalisation

- Replace the address by a tag that identifies the function.
- Use an algebraic data type to exclude invalid pairs `(tag, env)`: The pair becomes a constructor application `Ctag(env)`.

Example

```
...
let f = C1(a, b) in
let g = if a < b then f else C2() in
print (apply(f, 4) + apply(g, 5))


fun apply1((a, b), x) = a * x + b
fun apply2((), x) = x + 3

fun apply(g, x) = case g of
                    | C1(a, b) → apply1((a, b), x)
                    | C2() → apply2((), x)
```

Defunctionalisation (using flow information)

- Use control-flow information to improve the choice of data types.

Example

```
...
let f = fun x → a * x + b in
let g = if a < b then f else fun x → x + 3 in
print ((f 4) + (g 5))
```

Defunctionalisation (using flow information)

- Use control-flow information to improve the choice of data types.

Example

```
...
let f = (a, b) in
let g = if a < b then Left(f) else Right() in
print (apply1(f, 4) + apply(g, 5))

fun apply1((a, b), x) = a * x + b
fun apply2((), x) = x + 3

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

COMPOSITIONALITY & MODULARITY

Modularity and compositionality are very important in practice, but are often treated in an ad hoc way.

Example: Separate compilation

```
external
  f: ((int → int) → int) → int
  g: (int → int) → int
in f g end
```

Heap-allocated closures

Defunctionalisation

- `f` needs access to the `apply`-code in `g`, and vice versa
- closure repesentation in `g` may depend on `f`, and vice versa
- may need some new code at link time

MODULAR CLOSURE CONVERSION

Use a module system to formulate closure conversion!

- interface specification
- program composition
- basis for reasoning
- mathematical structure

FROM CALL-BY-VALUE PCF TO MODULES

CALL-BY-VALUE PCF

Use call-by-value PCF as a simple source language.

- simply-typed λ-calculus
- natural numbers with addition, case distinction, …
- recursion

$$
\begin{array}{rcl}
X, Y & ::= & \mathbb{N} \mid X \to Y \\
s, t & ::= & x \mid \lambda x{:}X.\, t \mid s\ t \\
& & \mid n \mid s + t \mid \texttt{if } s\texttt{=0 then } t_1 \texttt{ else } t_2 \\
& & \mid \texttt{fix}(t)
\end{array}
$$

TRANSLATION TO MODULES

Translate PCF to modules that contain only first-order functions.

Use an SML-like module system:

Structures

```
struct
    type t = int
    fun f (x: int) = x
end
```

Signatures

```
sig
  type t
  f: int → t
end
```

Functor

```
functor (X: S) =
  struct
     fun g (x: int) = X.f x
  end
```

Functor signature

```
functor (X: S) →
  sig
     g: int → X.t
  end
```

MODULAR CLOSURE CONVERSION

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

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

MODULAR CLOSURE CONVERSION

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

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

Base Type

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

MODULAR CLOSURE CONVERSION

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

$\mathcal{M}[\![X]\!]$:= sig
  include $\mathcal{I}[\![X]\!]$
  eval: unit → t
end

Base Type

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

Function type (special case)

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

MODULAR CLOSURE CONVERSION

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

$\mathcal{M}[\![X]\!]$:=
```
sig
  include 𝓘[[X]]
  eval: unit → t
end
```

Base Type

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

Function type (special case)

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

MODULAR CLOSURE CONVERSION

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

```
M⟦X⟧  :=  sig
            include I⟦X⟧
            eval: unit → t
          end
```

Base Type

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

Function type (special case)

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

MODULAR CLOSURE CONVERSION

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

```
M⟦X⟧  :=  sig
            include I⟦X⟧
            eval: unit → t
          end
```

Base Type

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

Function types (general case)

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

MODULAR CLOSURE CONVERSION

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

```
functor M_t (X1: I[[X_1]]) ... (Xn: I[[X_n]]) : sig
  include I[[Y]]
  eval: X1.t * ... * Xn.t → t
end
```

EXAMPLE — DEFUNCTIONALISATION

$\vdash \lambda y.\, y + 3 : \mathbb{N} \to \mathbb{N}$

translates to

```
functor M1 () = struct
  type t = unit

  fun eval() = ()

  functor T(Y: 𝓘⟦ℕ⟧) = struct

    structure T: 𝓘⟦ℕ⟧ = struct type t = int end

    fun apply ((f, y) : t * Y.t) : T.t =
      y + 3
  end
end
```

EXAMPLE — DEFUNCTIONALISATION

$x\colon (\mathbb{N} \to \mathbb{N}) \vdash \lambda y.\, x\ (y+1) :\ \mathbb{N} \to \mathbb{N}$

translates to

```
functor M₂ (X: ℐ⟦ℕ → ℕ⟧) = struct
  type t = X.t

  fun eval(x: X.t) : t = x

  functor T(Y: ℐ⟦ℕ⟧) = struct

    structure T: ℐ⟦ℕ⟧ = struct type t = int end

    fun apply ((f, y) : t * Y.t) : T.t =
      X.T(Y).apply(f, y+1)
  end
end
```

EXAMPLE — DEFUNCTIONALISATION

$x{:}(\mathbb{N} \to \mathbb{N}) \vdash$ `if` $(x\ 2) = 0$ `then` $\lambda y.\, y + 1$ `else` $\lambda y.\, x\ (y + 3) : \mathbb{N} \to \mathbb{N}$

translates to

```
functor M3 (X: I[[N -> N]]) = struct
  datatype t = Left of unit | Right of X.t

  fun eval(x : X.t) : t =
    if X.T(struct type t = int end).apply(x, 2) = 0
      then Left() else Right(x)

  functor T(Y: I[[N]]) = struct
    structure T: I[[N]] = struct type t = int end
    fun apply ((f, y) : t * Y.t) : T.t =
      case f of Left() => y + 1
              | Right(x) => X.T(Y).apply(x, y+3)
  end
end
```

MODULAR CLOSURE CONVERSION

It is not hard to define a translation for all of call-by-value PCF by induction on typing derivations.

- Abstraction of data types like in typed closure conversion [Minamide, Harper, Morrisett]
- Abstraction of application functions: in principle, each λ-abstraction could use a different closure repesentation
- Linking of separately translated terms immediate.

What have we gained?

FRAGMENTS OF LOW-LEVEL CODE AS MODULES

LOW-LEVEL PROGRAMS

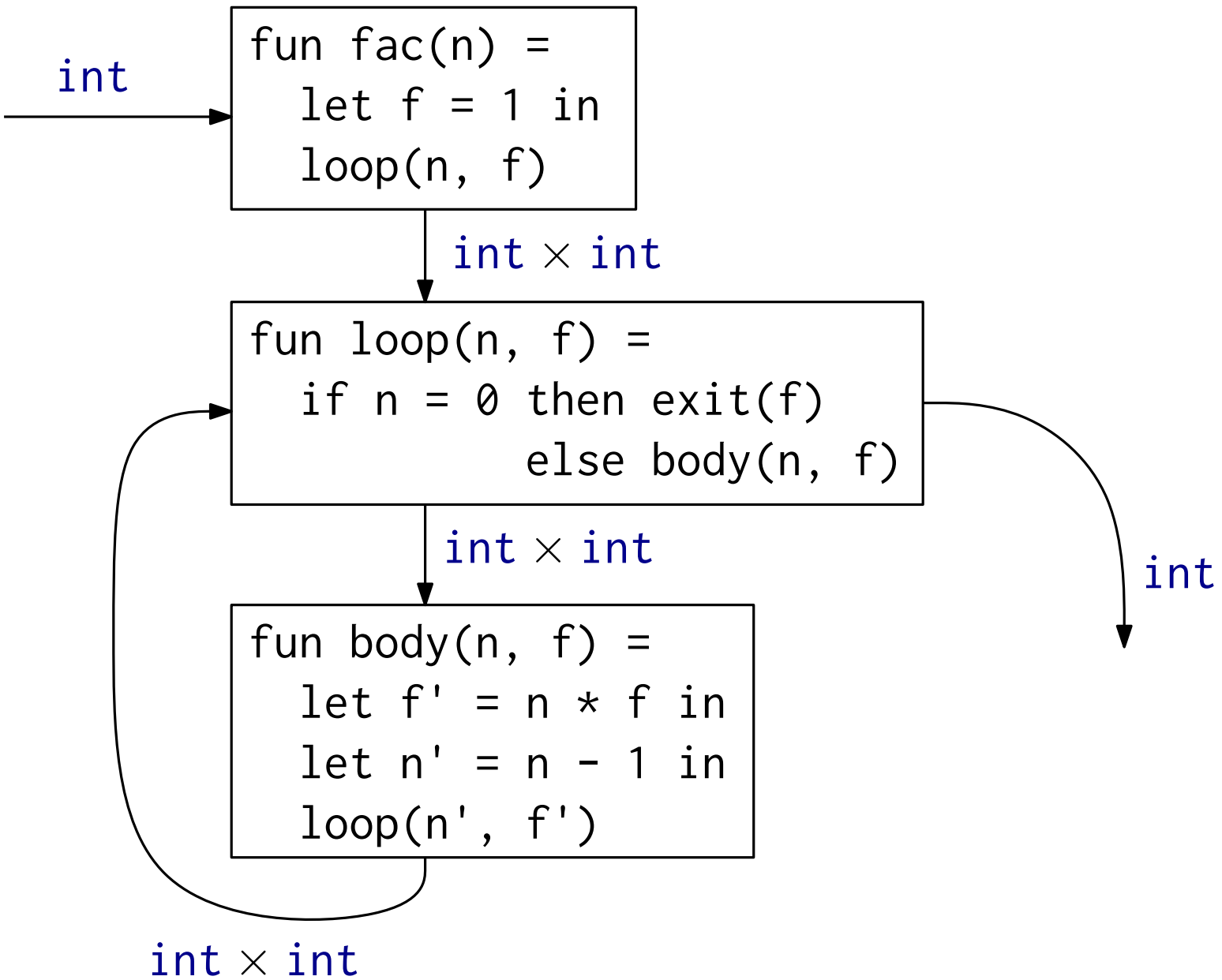
Low-level programs

- set of function definitions (blocks)
- first-order data
- tail calls only

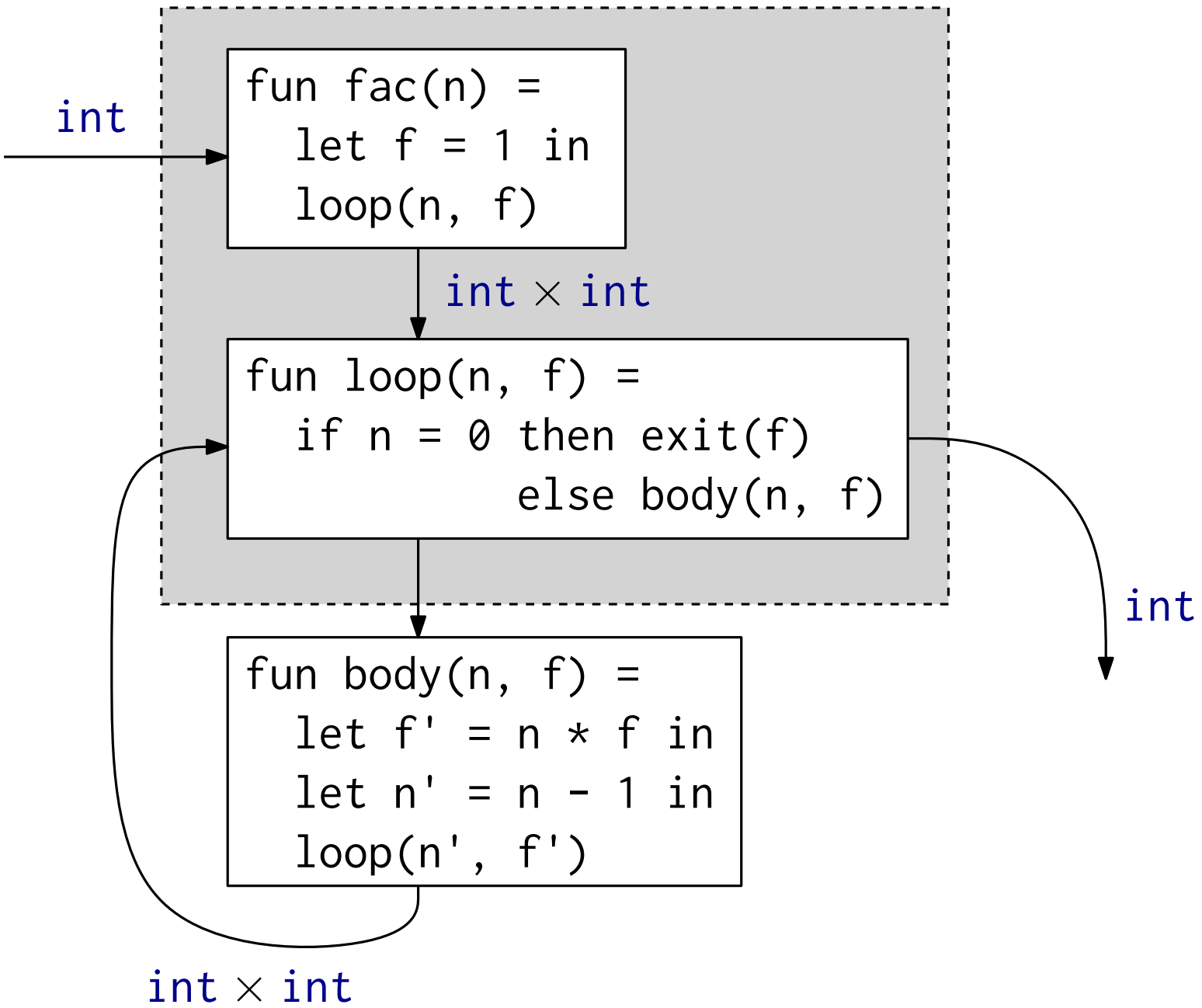
Example:

```
fun fac(n) =
  let f = 1 in
  loop(n, f)

fun loop(n, f) =
  if n = 0 then ret(f)
           else body(n, f)

fun body(n, f) =
  let f' = n * f in
  let n' = n - 1 in
  loop(n', f')
```

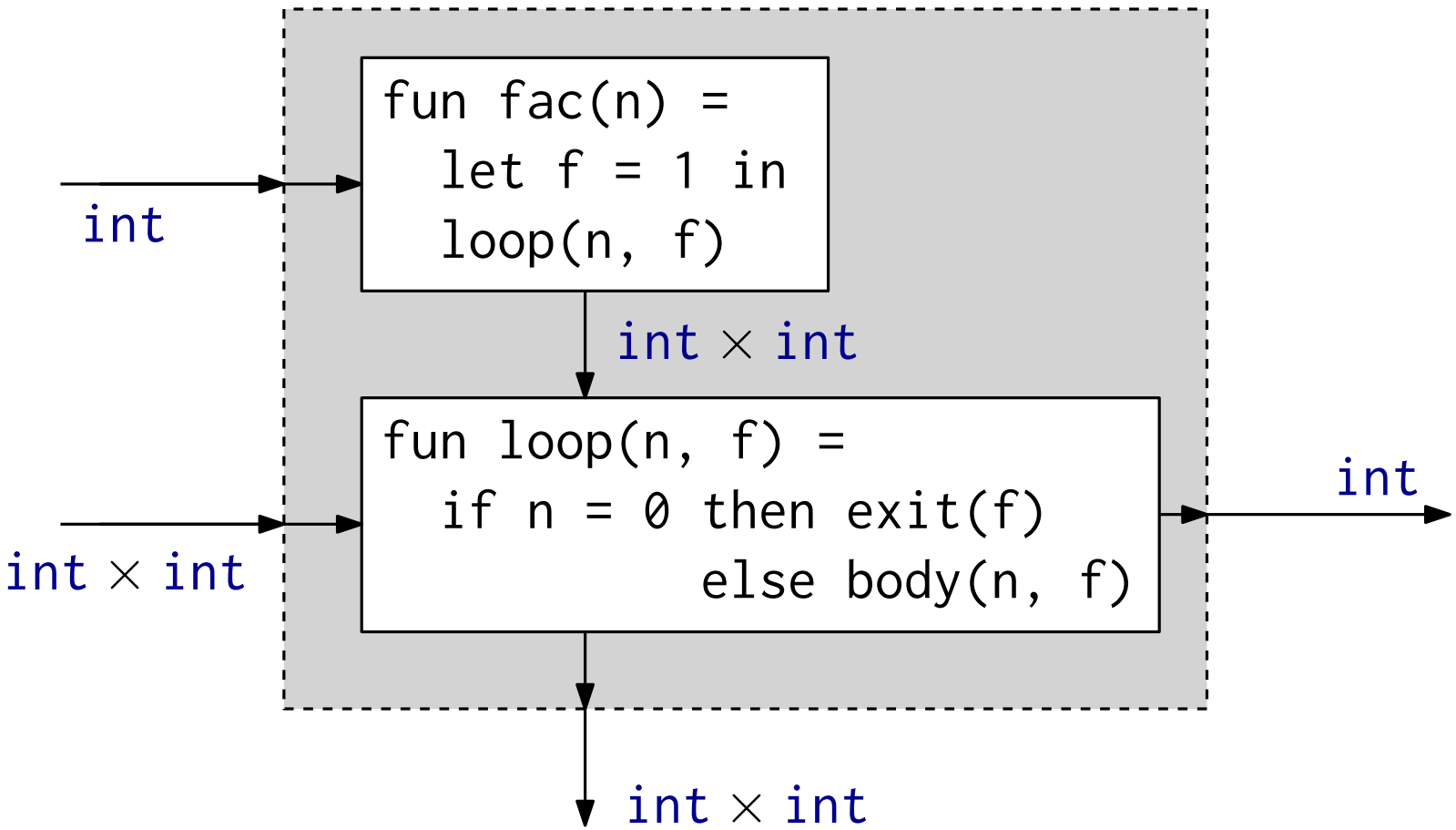
CONTROL FLOW GRAPHS



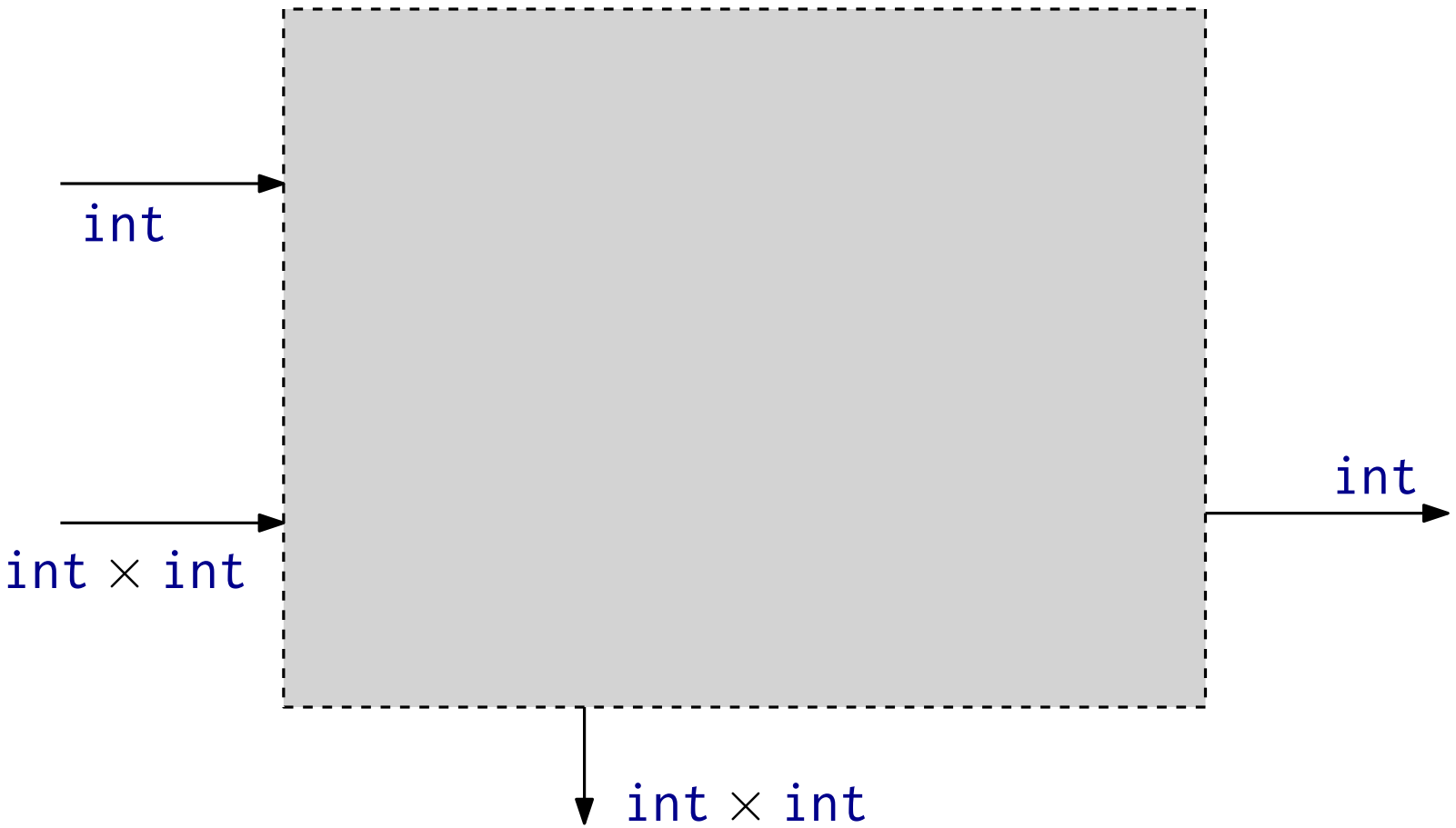
PROGRAM FRAGMENT



PROGRAM FRAGMENT



PROGRAM FRAGMENT



FRAGMENTS AS LITTLE MODULES

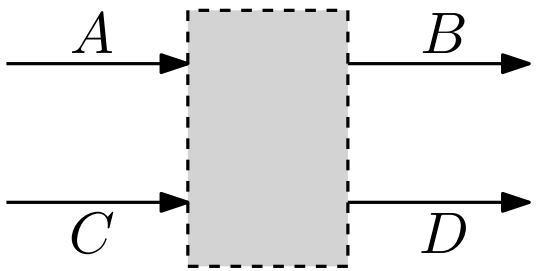
Consider low-level program fragments as modules that compose into the final program.

Specify their interfaces using ML-style signatures.

Example:

```
sig
  f: A → B
  g: C → D
end
```

is implemented by



FRAGMENTS AND THEIR SIGNATURES

Signatures

$$
\begin{aligned}
S \;::=\; & A \to B \\
\mid\; & \texttt{sig}\; X{:}S, \ldots, X{:}S\; \texttt{end} \\
\mid\; & \texttt{functor}(X{:}S) \to S \\
\mid\; & A \cdot S \\
\mid\; & \forall \alpha \lhd A.\, S \\
\mid\; & \exists \alpha \lhd A.\, S
\end{aligned}
$$

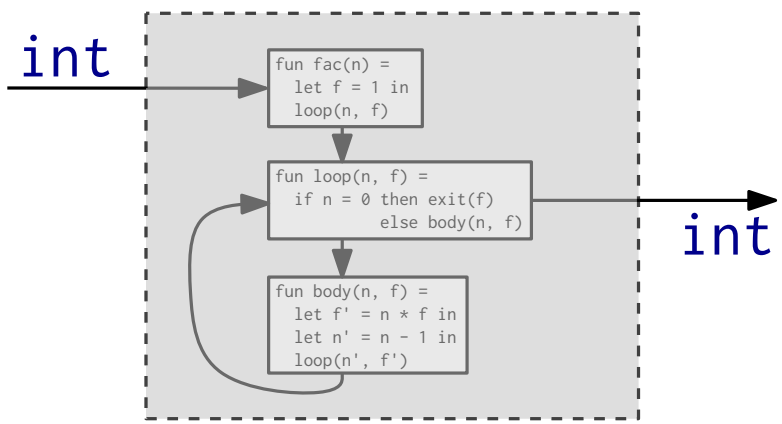
A signature specifies interface and (some) behaviour of program fragments.

FRAGMENTS AND THEIR SIGNATURES

- Signature $A \to B$ is implemented by fragments of interface:

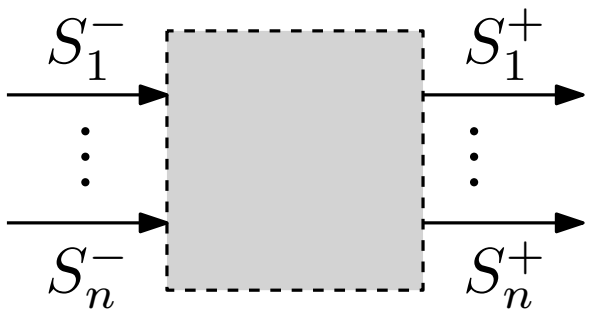
Example:

```
sig
  fac: int → int
end
```



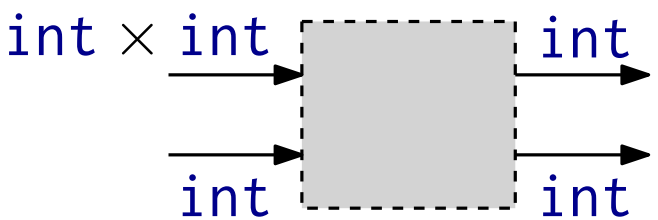
FRAGMENTS AND THEIR SIGNATURES

- `sig` $X_1{:}S_1, \ldots, X_n{:}S_n$ `end`



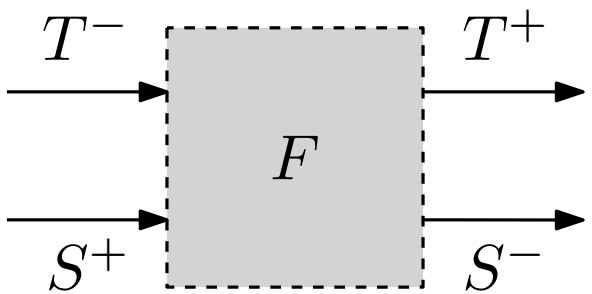
Example:

```
sig
  add: int × int → int
  square: int → int
end
```

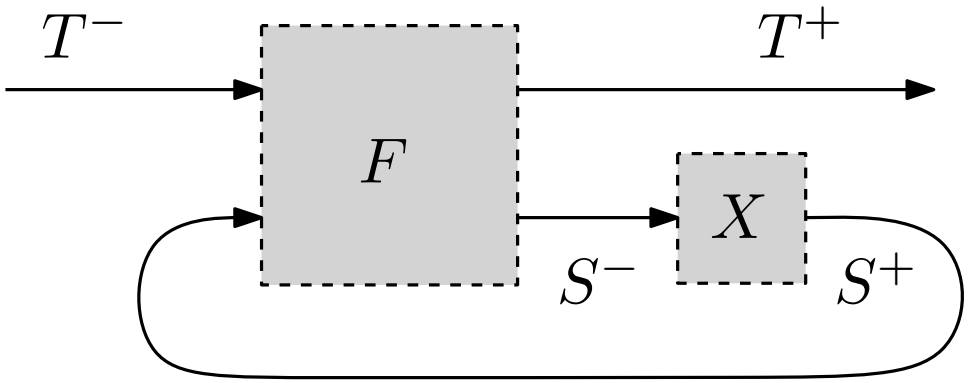


FRAGMENTS AND THEIR SIGNATURES

- `functor`$(X{:}S) \to T$



Intention:

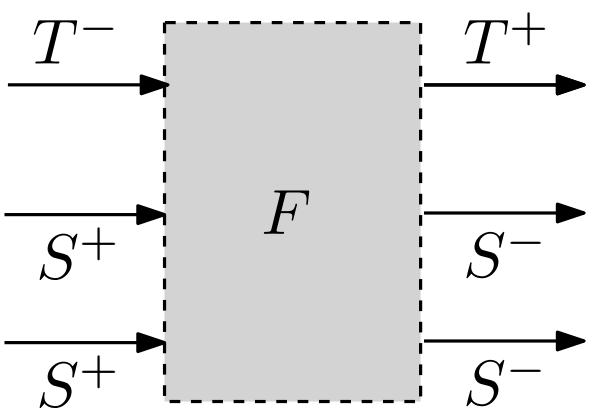


FRAGMENTS AND THEIR SIGNATURES

- $A \cdot S$ (*subexponential*)

The first argument is returned unchanged and uninspected.

Example: Contraction

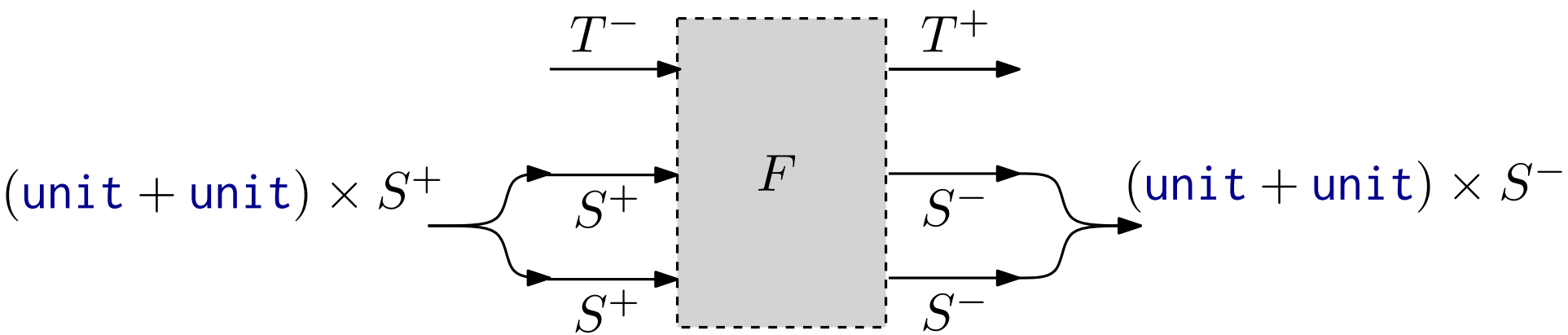


FRAGMENTS AND THEIR SIGNATURES

- $A \cdot S$ (*subexponential*)

The first argument is returned unchanged and uninspected.

Example: Contraction

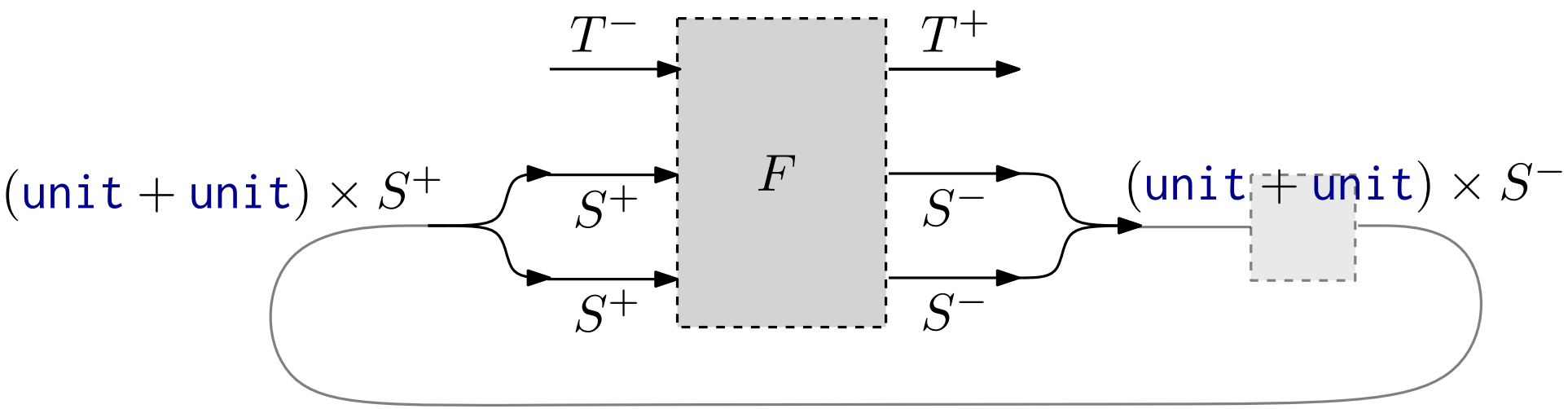


FRAGMENTS AND THEIR SIGNATURES

- $A \cdot S$ (*subexponential*)

The first argument is returned unchanged and uninspected.

Example: Contraction



FRAGMENTS AS MODULES

The SML module system can be translated to System F_ω [F-ing Modules: Rosberg, Russo, Dreyer].

Our low-level interpretation of modules can be understood as a variant of this translation to INT, a linear variant of System F.

FROM ANNOTATED PCF TO LOW-LEVEL CODE

TRANSLATION TO LOW-LEVEL PROGRAMS

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

```
M⟦X⟧  :=  sig
            include I⟦X⟧,
            eval: unit → T.t
          end
```

Base Type

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

Function types

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

TRANSLATION TO LOW-LEVEL PROGRAMS

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

```
M⟦X⟧  :=  sig
            include I⟦X⟧,
            eval: A · (unit → T.t)
          end
```

Base Type

```
I⟦ℕ⟧ = A · (sig
    type◁int t = int
end)
```

Function types

```
I⟦X → Y⟧ = A · (sig
  type◁C t
  T: functor (X: I⟦X⟧) → sig
        T : I⟦Y⟧
        apply: B · (t × X.t → T.t)
      end
end)
```

TRANSLATION TO LOW-LEVEL PROGRAMS

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

$$\mathcal{M}[\![X]\!] \quad := \quad \texttt{sig}$$
$$\texttt{include } \mathcal{I}[\![X]\!]$$
$$\texttt{eval: } A \cdot (\texttt{unit} \to \texttt{t})$$
$$\texttt{end}$$

Base Type

$$\mathcal{I}[\![A \cdot \mathbb{N}]\!] = A \cdot (\texttt{sig}$$
$$\quad \texttt{type t = int}$$
$$\texttt{end})$$

Function types (general case)

$$\mathcal{I}[\![A \cdot (X \xrightarrow{C}_B Y)]\!] = A \cdot (\texttt{sig}$$
$$\texttt{type} \triangleleft C \ \texttt{t}$$
$$\texttt{T: functor (X: } \mathcal{I}[\![X]\!]) \to \texttt{sig}$$
$$\texttt{T : } \mathcal{I}[\![Y]\!]$$
$$\texttt{apply: } B \cdot (\texttt{t} \times \texttt{X.t} \to \texttt{T.t})$$
$$\texttt{end}$$
$$\texttt{end})$$

TYPE SYSTEM TRACKS ANNOTATIONS

$$\textsf{APP}\ \frac{\vdash_U s : \mathbb{N} \xrightarrow{C}_U \mathbb{N} \qquad \vdash_{U \times C} t : \mathbb{N}}{\vdash_U s\ t : \mathbb{N}}$$

amounts to

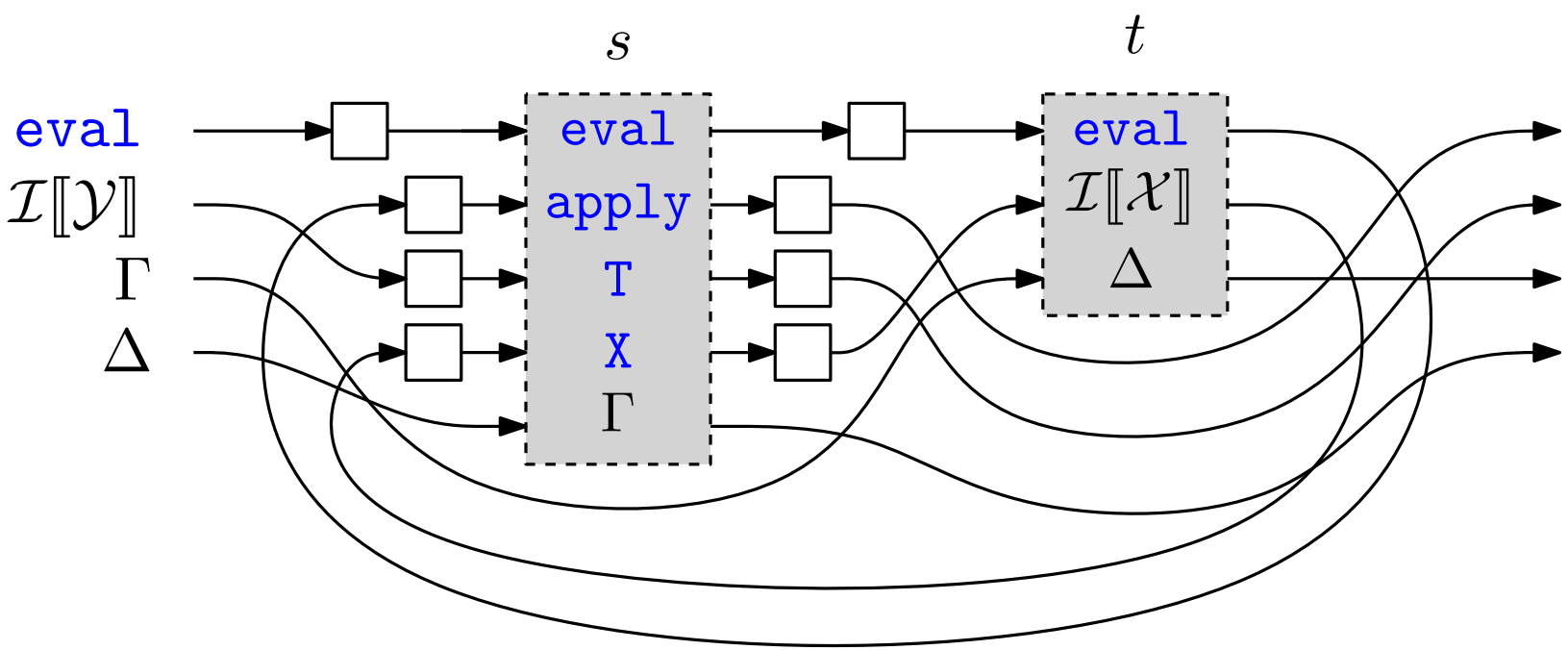
s: sig
  type$_{\triangleleft C}$ t
  apply: $U \cdot$(t $\times$ int $\to$ int)
  eval: $U \cdot$(unit $\to$ t)
end

t: sig
  type t = int
  eval: $(U \times C) \cdot$(unit $\to$ int)
end

$s\ t$: sig
  type t = int
  eval: $U \cdot$(unit $\to$ int)
end

TYPE SYSTEM TRACKS ANNOTATIONS

$$\textsf{APP}\ \frac{\Gamma \vdash_{U \times C[\![\Delta]\!]} s : \mathcal{X} \xrightarrow{C}_U \mathcal{Y} \qquad \Delta \vdash_{U \times C} t : \mathcal{X}}{\Gamma, \Delta \vdash_U s\ t : \mathcal{Y}}$$



WHAT HAVE WE GAINED?

Modular formulation of defunctionalisation

- specification of low-level interfaces
- flexible choice of low-level details
  (represented by type annotations)
- whole-program analysis becomes type inference

The proof of correctness factors in two manageable parts.

1. Correctness of translation to modules
2. Correctness of low-level implementation of modules

THANK YOU FOR YOUR ATTENTION!