Interaction Semantics and Programming Language Compilation

Ulrich Schöpp
LMU Munich

Introduction

Interaction Semantics builds mathematical models for programming languages from interacting processes.

Such models can help understand low-level decompositions of high-level languages.

```
let rec fib x =
     if x < 1 then 1 else (fib (x - 1)) + (fib (x - 2))
```

```
...
%x6 = phi i32 [ 38, %case1 ], [ %unpack35, %case145 ],
                [ %add, %case167 ]
%add = add i32 %x6, -1
%eq47 = icmp ne i32 %add, 0
switch i1 %eq47, label %case049 [ i1 true, label %case148 ]
...
```

Introduction

Need better understanding for:

- formal verification
- compositional reasoning
- resource usage analysis and certification
- modularity

Game Semantics for Logic

Explain logic in terms of dialogues between disputing parties.

Proponent and Opponent argue about a proposition:

- Proponent tries to defend it.
- Opponent tries to refute it.
- The logic defines the mode of interaction.
  - How can a formula be attacked?
  - How can a formula be defended?

A proof is a strategy for Proponent to defend the proposition against any possible attack.

Game Semantics for Constructive Logic

[Lorenzen & Lorenz, 1950s]

$$(\bot \wedge \varphi) \vee \top$$

Opponent **Proponent**

Which of the disjuncts is true?

Game Semantics for Constructive Logic

[Lorenzen & Lorenz, 1950s]

$$(\bot \wedge \varphi) \vee \top$$

Opponent

Proponent

Which of the disjuncts is true?

The left one $\bot \wedge \varphi$ *is true.*

Game Semantics for Constructive Logic

[Lorenzen & Lorenz, 1950s]

$$(\bot \wedge \varphi) \vee \top$$

Opponent

Proponent

Which of the disjuncts is true?

The left one $\bot \wedge \varphi$ *is true.*

Then explain why $\bot$ *is true.*

Game Semantics for Programs

Proponent now defends the claim:

I have a program of type X.

Attacks become requests for information.

Programs are modelled by strategies that explain how Proponent can answer any request for information.

Game Semantics for Programs

$\texttt{int} \to \texttt{int}$

Opponent | **Proponent**

What does your function return?

Game Semantics for Programs

$$\texttt{int} \rightarrow \texttt{int}$$

Opponent | **Proponent**

What does your function return?

What is the function argument?

Game Semantics for Programs

$\texttt{int} \rightarrow \texttt{int}$

Opponent | **Proponent**

What does your function return?

What is the function argument?

The argument is 5.

Game Semantics for Programs

$$\texttt{int} \to \texttt{int}$$

Opponent

Proponent

What does your function return?

What is the function argument?

The argument is 5.

Then the function returns 6.

Game Semantics for Programs

The strategy of a program derives from strategies of its parts.

$$\lambda x.\, x + 1 \colon \texttt{int} \to \texttt{int}$$

$$x + 1 \colon \texttt{int}$$

$$x \colon \texttt{int} \qquad\qquad 1 \colon \texttt{int}$$

Game Semantics for Programs

The strategy of a program derives from strategies of its parts.

What does the function return?

$$\lambda x.\, x + 1 \colon \texttt{int} \to \texttt{int}$$

$$x + 1 \colon \texttt{int}$$

$x \colon \texttt{int}$ $\qquad$ $1 \colon \texttt{int}$

Game Semantics for Programs

The strategy of a program derives from strategies of its parts.

What does the function return?

$$\lambda x.\, x + 1 \colon \texttt{int} \to \texttt{int}$$

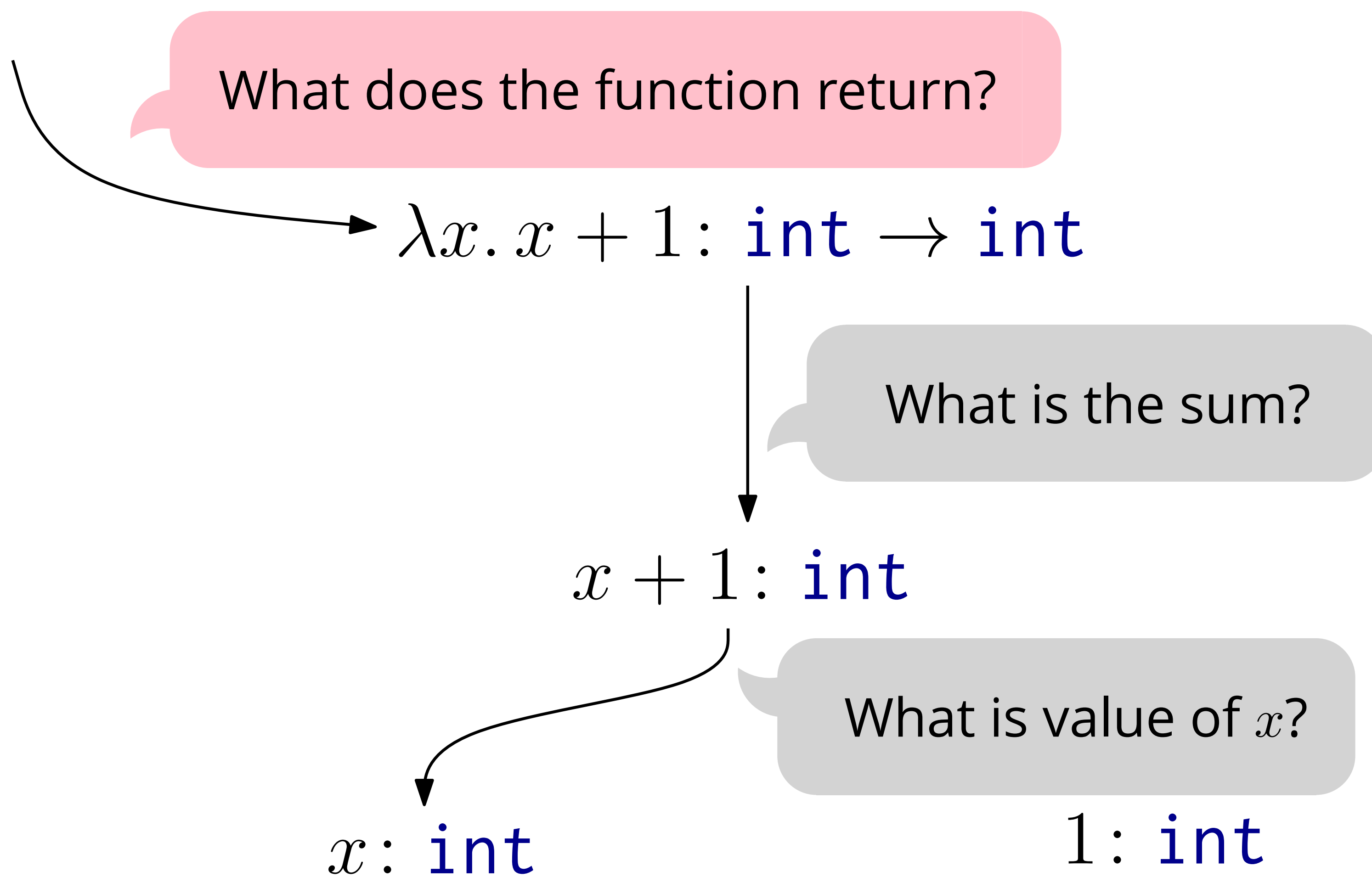
What is the sum?

$$x + 1 \colon \texttt{int}$$

$x : \texttt{int}$ $\qquad$ $1 \colon \texttt{int}$

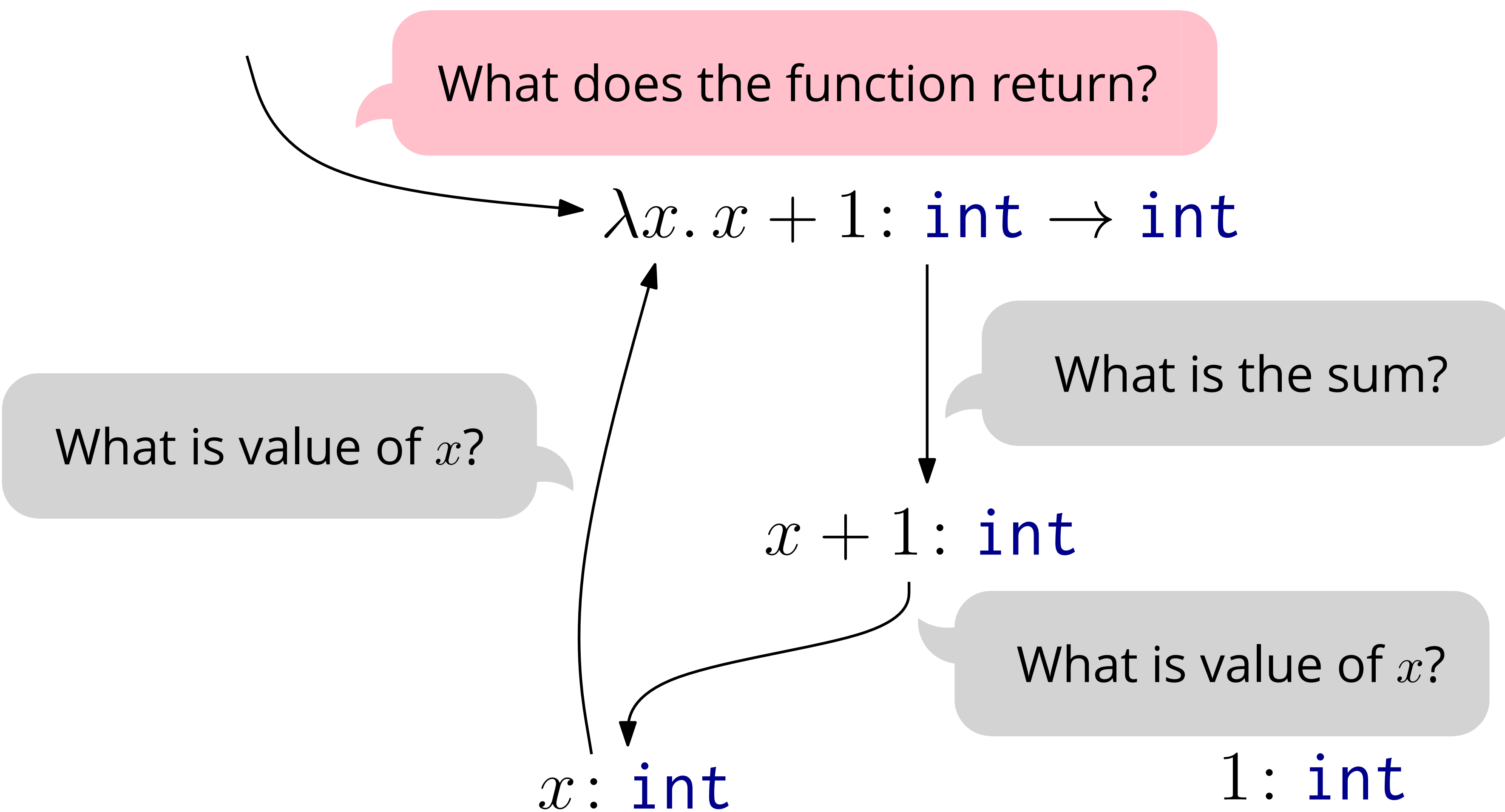
Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



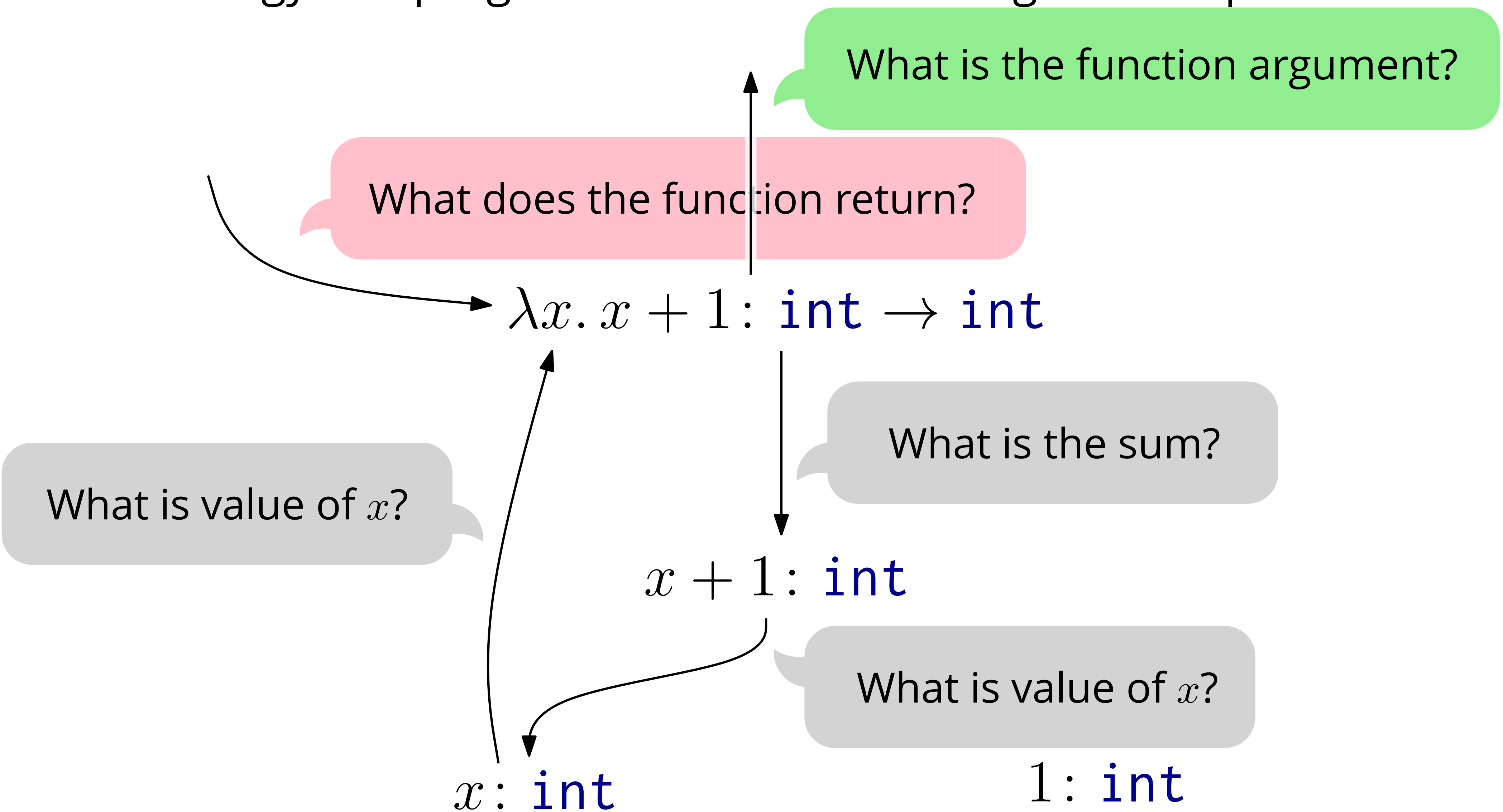
Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



Game Semantics for Programs

The strategy of a program derives from strategies of its parts.

The argument is 5.

$$\lambda x.\, x + 1 \colon \texttt{int} \to \texttt{int}$$

$$x + 1 \colon \texttt{int}$$

$$x \colon \texttt{int} \qquad 1 \colon \texttt{int}$$

Game Semantics for Programs

The strategy of a program derives from strategies of its parts.

The argument is 5.

$$\lambda x.\, x + 1 \colon \texttt{int} \to \texttt{int}$$

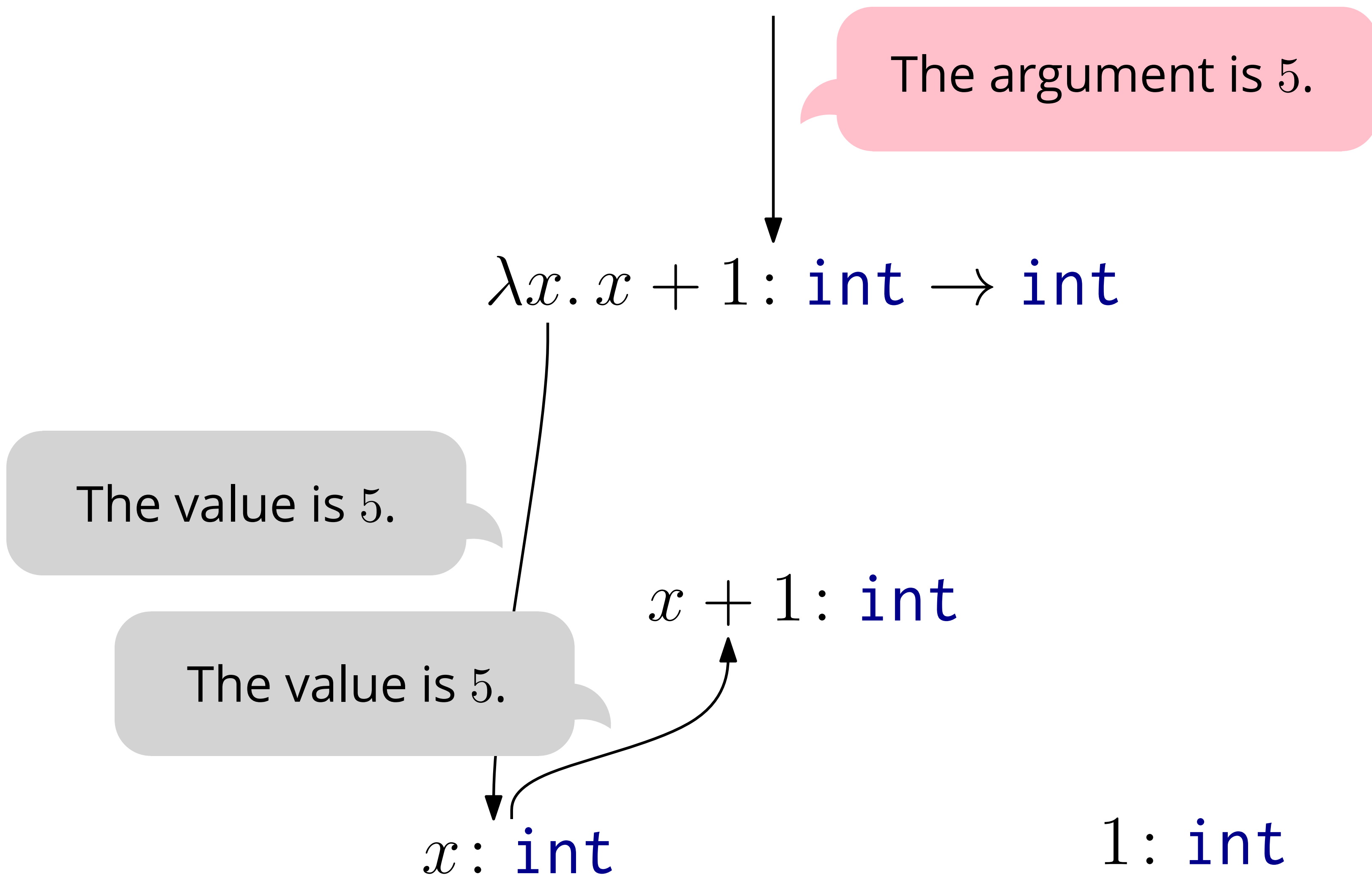
The value is 5.

$$x + 1 \colon \texttt{int}$$

$$x \colon \texttt{int} \qquad\qquad 1 \colon \texttt{int}$$

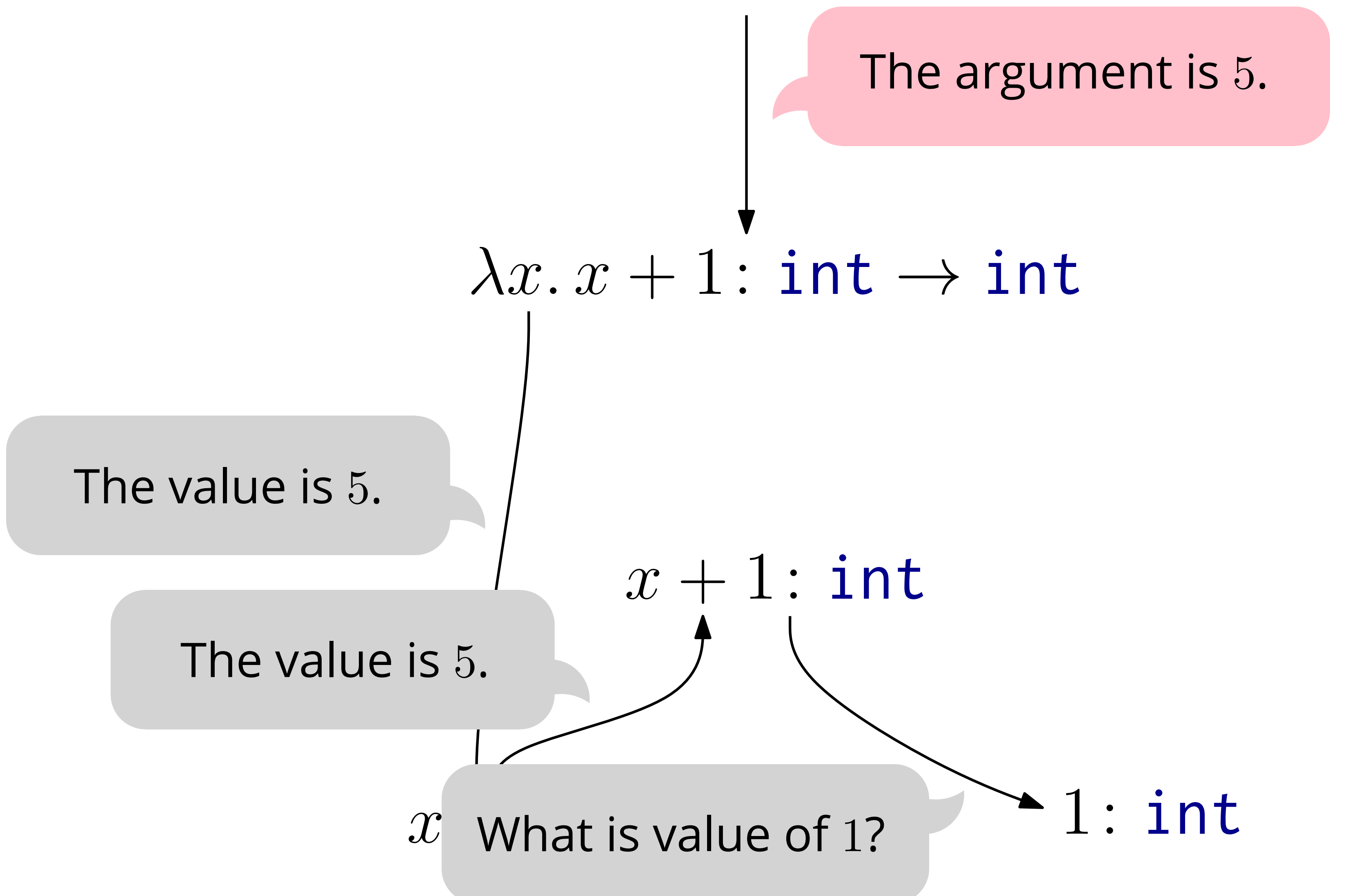
Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



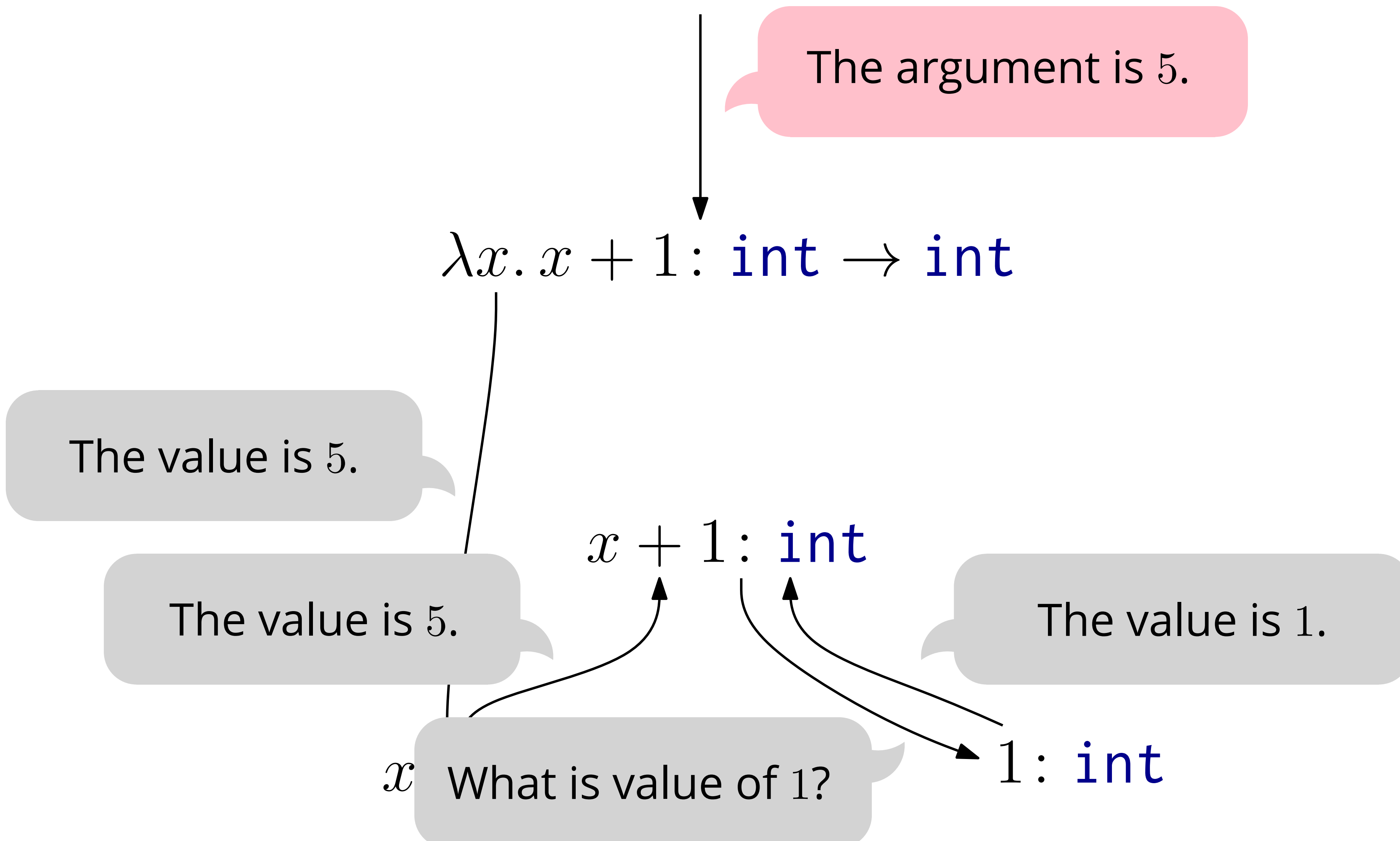
Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



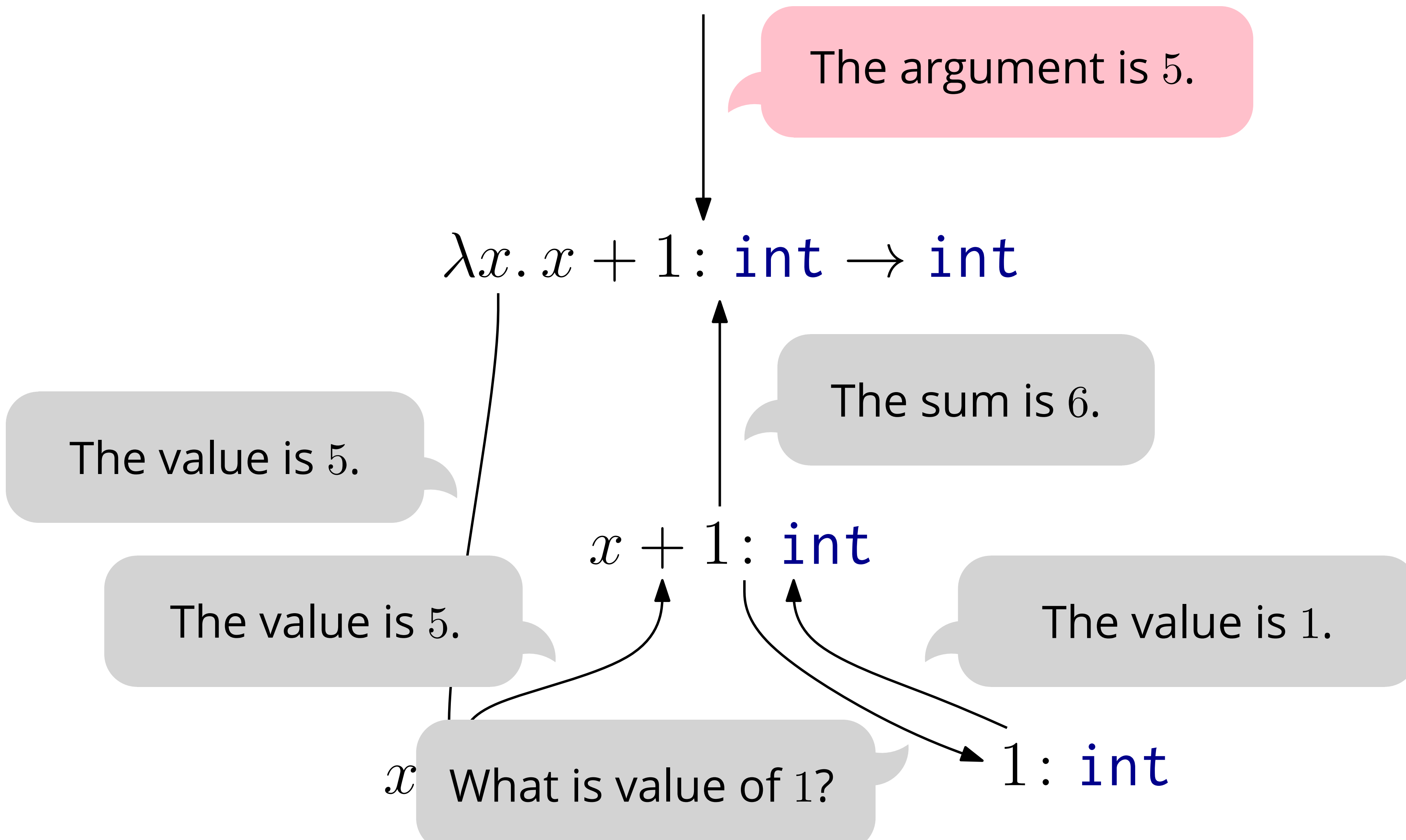
Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



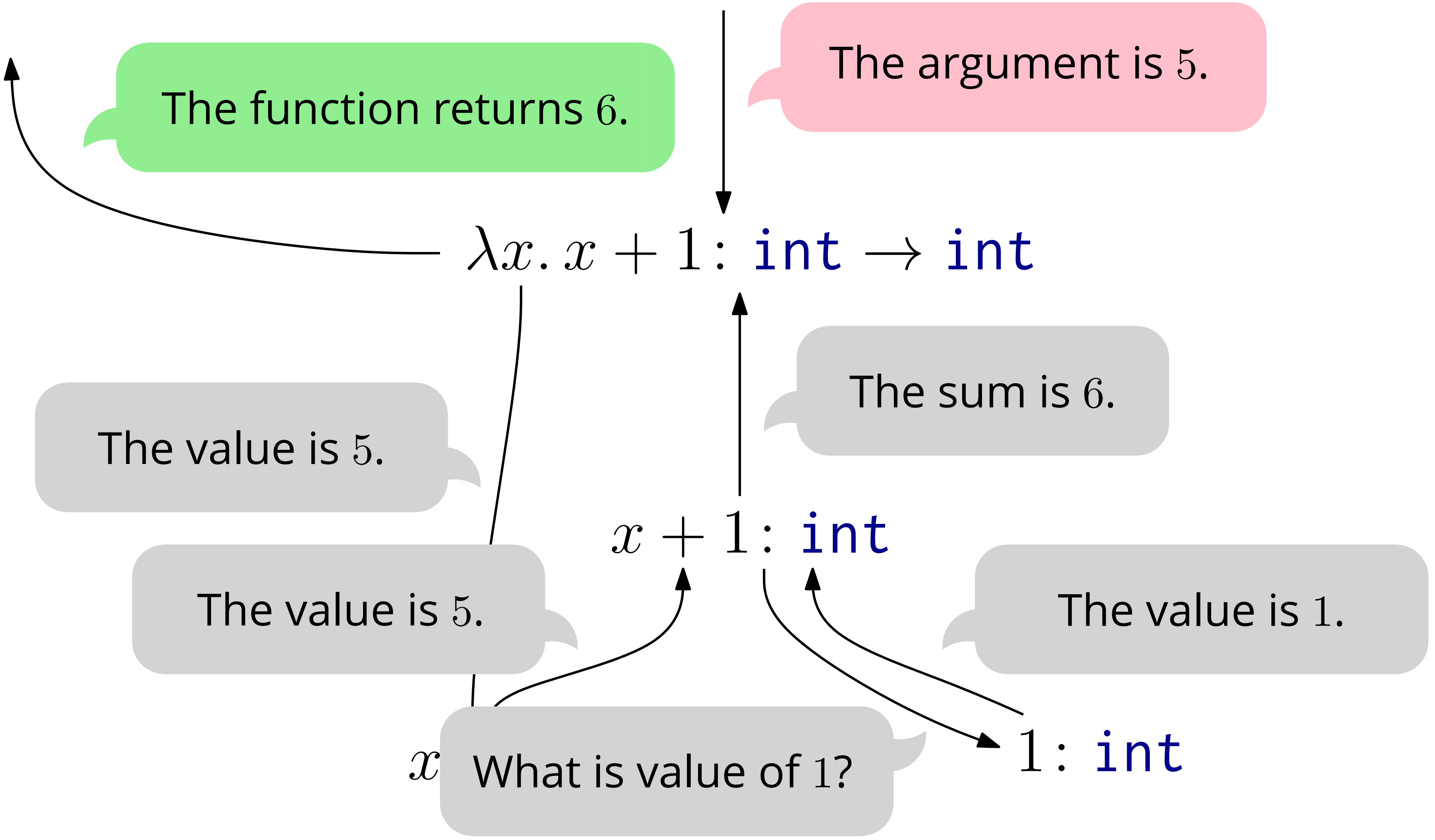
Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



Game Semantics for Programs

The strategy of a program derives from strategies of its parts.



Structure in Game Semantics

Game semantics has developed a number of mathematical constructions that turn a very simple model of interaction dialogues into precise models of many programming languages.

Fully abstract model for PCF

- [Hyland & Ong, 1994]
- [Abramsky, Jagadeesan & Malacaria, 1994]
- [Nickau 1994]

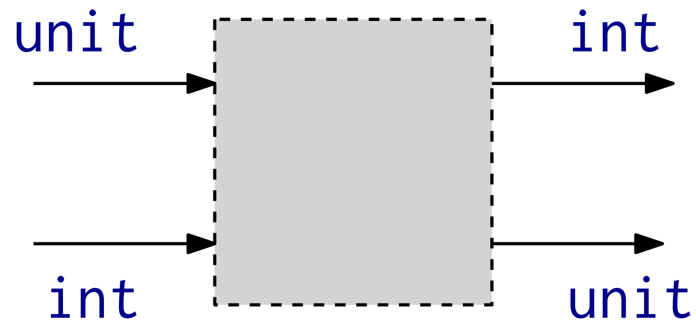
Geometry of Interaction

- closely related, with proof-theoretic motivation [Girard 1987]

Computation by Interaction

Implement programs by implementing their interaction strategies.

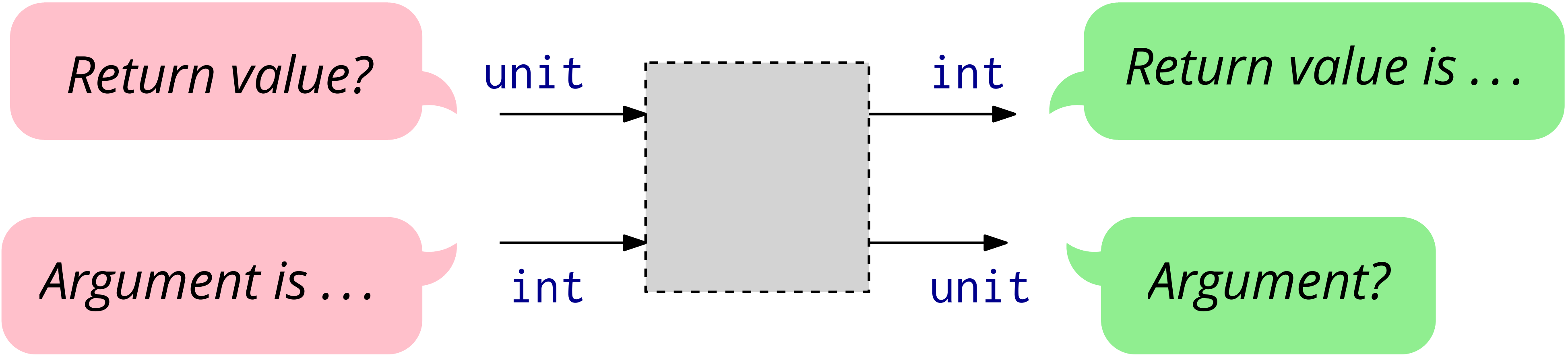
$$\texttt{int} \to \texttt{int}$$



Computation by Interaction

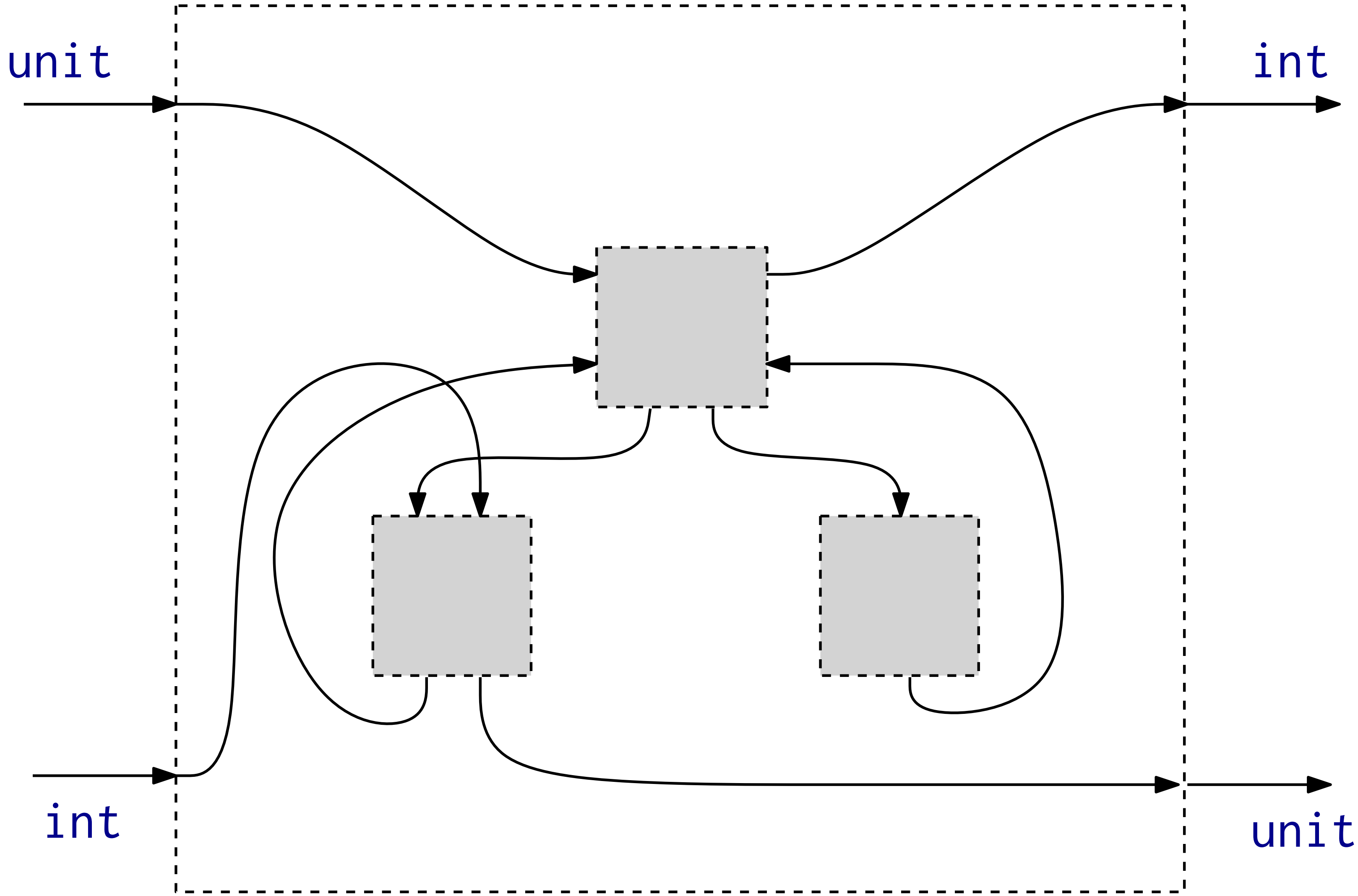
Implement programs by implementing their interaction strategies.

$$\texttt{int} \to \texttt{int}$$



Computation by Interaction

Strategies are compositional building blocks.



Computation by Interaction

Construct a game semantic model from low-level programs.
Interpretation becomes compilation.

Developed for compilation to . . .

- abstract machines [Mackie, 1995]
- hardware circuits [Ghica, Smith & Singh, 2007]
- LOGSPACE Turing Machines [S., 2006], [Dal Lago & S., 2010]
- π-calculus [Honda, Yoshida & Berger, 2001]
- distributed processes [Fredriksson & Ghica, 2013]
- quantum circuits
  [Hoshino, Hasuo, Yoshimizu, Faggian, Dal Lago, 2014]
- . . .

Introduction

Consider interaction as a general approach to connect mathematical semantics to compiler construction.

Semantics

- mathematical structure
- compositionality
- proofs

Compiler Construction

- efficiency
- optimisations
- implementation

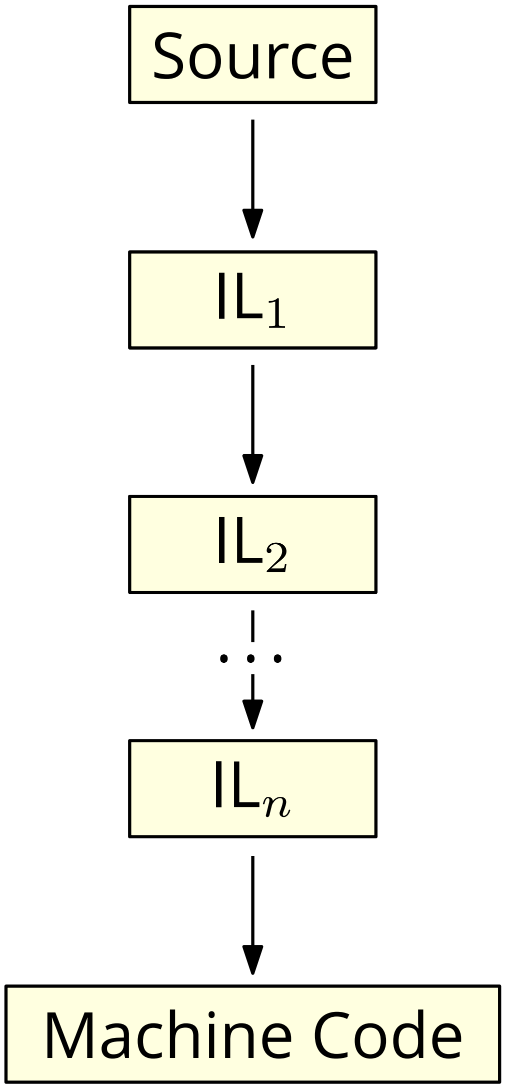
Overview

We look at the compilation of higher-order functional programming languages.

Compilers work by translating the source into a number of intermediate languages.

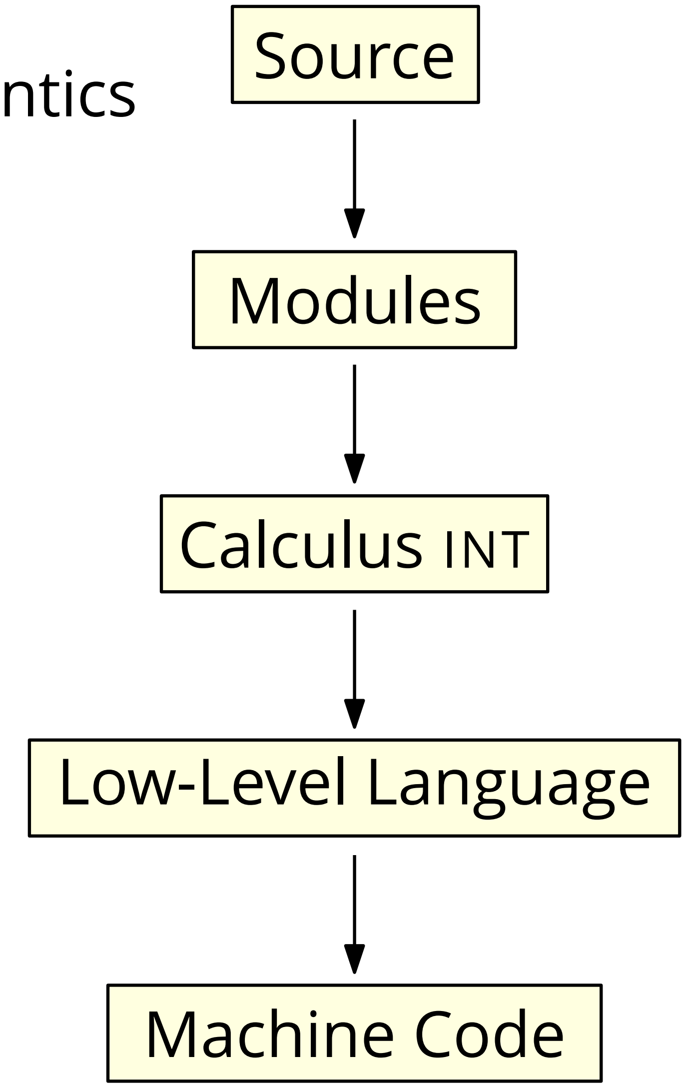
- decreasing level of abstraction
- optimisations at different levels

We use constructions from interaction semantics to construct a series of intermediate languages.



Overview

- Low-Level Programs
- Organising Low-Level Programs
  - Constructions from Interaction Semantics
  - Calculus INT
  - Simple Module System
- Compilation
  - Call-by-Name
  - Call-by-Value
- Relation to Defunctionalisation



Low-Level Programs

Low-Level Programs

Most compilers abstract from machine details by translating to an architecture-independent low-level language that is then translated to machine code.

Example: LLVM compiler infrastructure

- used by many compilers (Clang, Rust, . . .)
- portable assembler in static single assignment form (simple instructions, jumps, machine calls)
- compiler for many architectures

Low-Level Programs

LLVM IR

```
entry:
  ; initial value = 1.0 (inlined into phi)
  br label %loop

loop:       ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  ; body
  %calltmp = call double @putchard(double 4.200000e+01)
  ; increment
  %nextvar = fadd double %i, 1.000000e+00

  ; termination test
  %cmptmp = fcmp ult double %i, %n
  %booltmp = uitofp i1 %cmptmp to double
  %loopcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %loopcond, label %loop, label %afterloop

afterloop:      ; preds = %loop
  ; loop always returns 0.0
  ret double 0.000000e+00
```

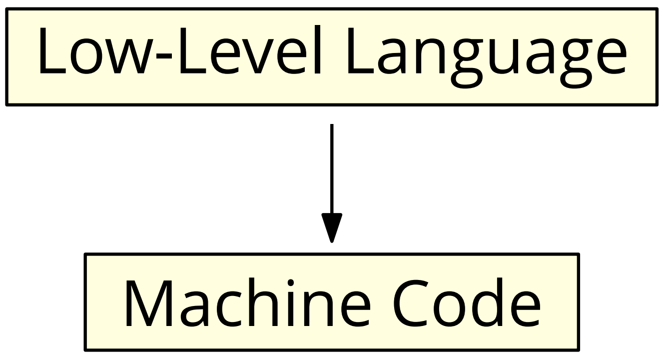
(Source: `http://llvm.org/docs/tutorial/LangImpl05.html`)

Low-Level Programs

We define a simple low-level language:

- similar abstraction level as LLVM assembly
- idealised heap (recursive types)
- functional presentation of static single assignment form

Similar languages are used in production compilers, e.g. Swift Intermediate Language.



Values and Types

Types

$$A, B \ ::= \ \alpha \mid \mathtt{int} \mid \mathtt{unit} \mid A \times B \mid 0 \mid A + B \mid \mu\alpha.A$$

Values

$$v, w ::= () \mid n \mid (v, w) \mid \mathtt{inl}(v) \mid \mathtt{inr}(v) \mid \mathtt{fold}(v)$$

Use algebraic data types as syntactic sugar for $\mu\alpha.A$.

Example: Write

$$\begin{aligned} \mathtt{type\ list}\langle\alpha\rangle \ &= \mathtt{Nil}\ \text{of}\ \mathtt{unit} \\ &\mid \text{Cons of}\ \alpha \times \mathtt{list}\langle\alpha\rangle \end{aligned}$$

for $\mu\beta.\,\mathtt{unit} + \alpha \times \beta$, where $\mathtt{Nil} = \mathtt{fold}(\mathtt{inl}())$ and $\text{Cons}(h, t) = \mathtt{fold}(\mathtt{inr}(h, t))$.

Blocks

Programs are constructed from blocks.

A **block** has the form

$$\texttt{label}(x : A) = \mathit{body}$$

where

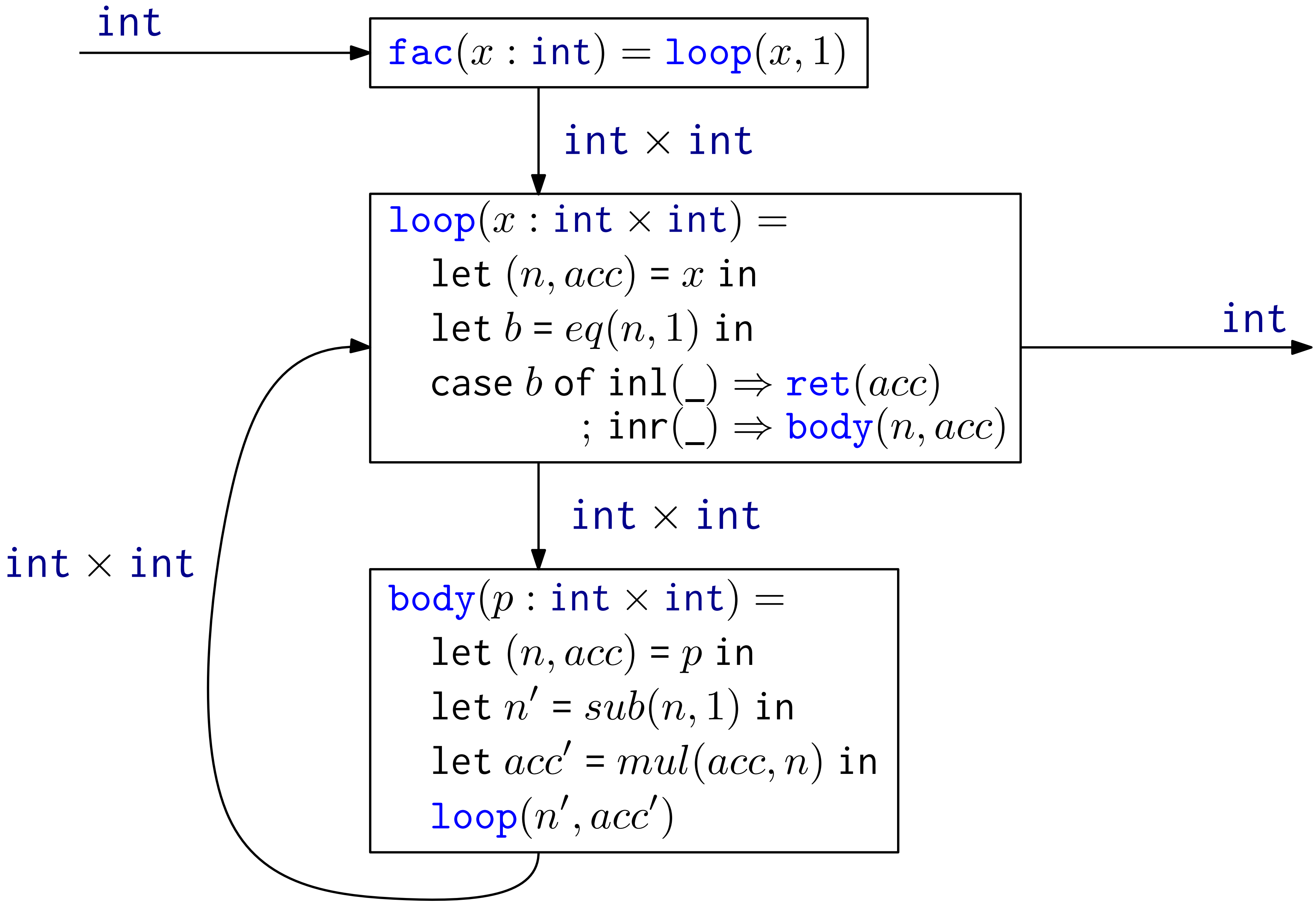
$$\begin{aligned}
\mathit{body} \;::=\; & \texttt{let } x = \texttt{primop}(v) \texttt{ in } \mathit{body} \\
\mid\; & \texttt{let } (x, y) = v \texttt{ in } \mathit{body} \\
\mid\; & \texttt{let fold}(x) = v \texttt{ in } \mathit{body} \\
\mid\; & \texttt{label}(v) \\
\mid\; & \texttt{case } v \texttt{ of inl}(x) \to \texttt{label}_1(v_1)\,;\ \texttt{inr}(y) \to \texttt{label}_2(v_2)
\end{aligned}$$

`primop` ranges over primitive operations, such as `add`, `mul`, or `syscall`,

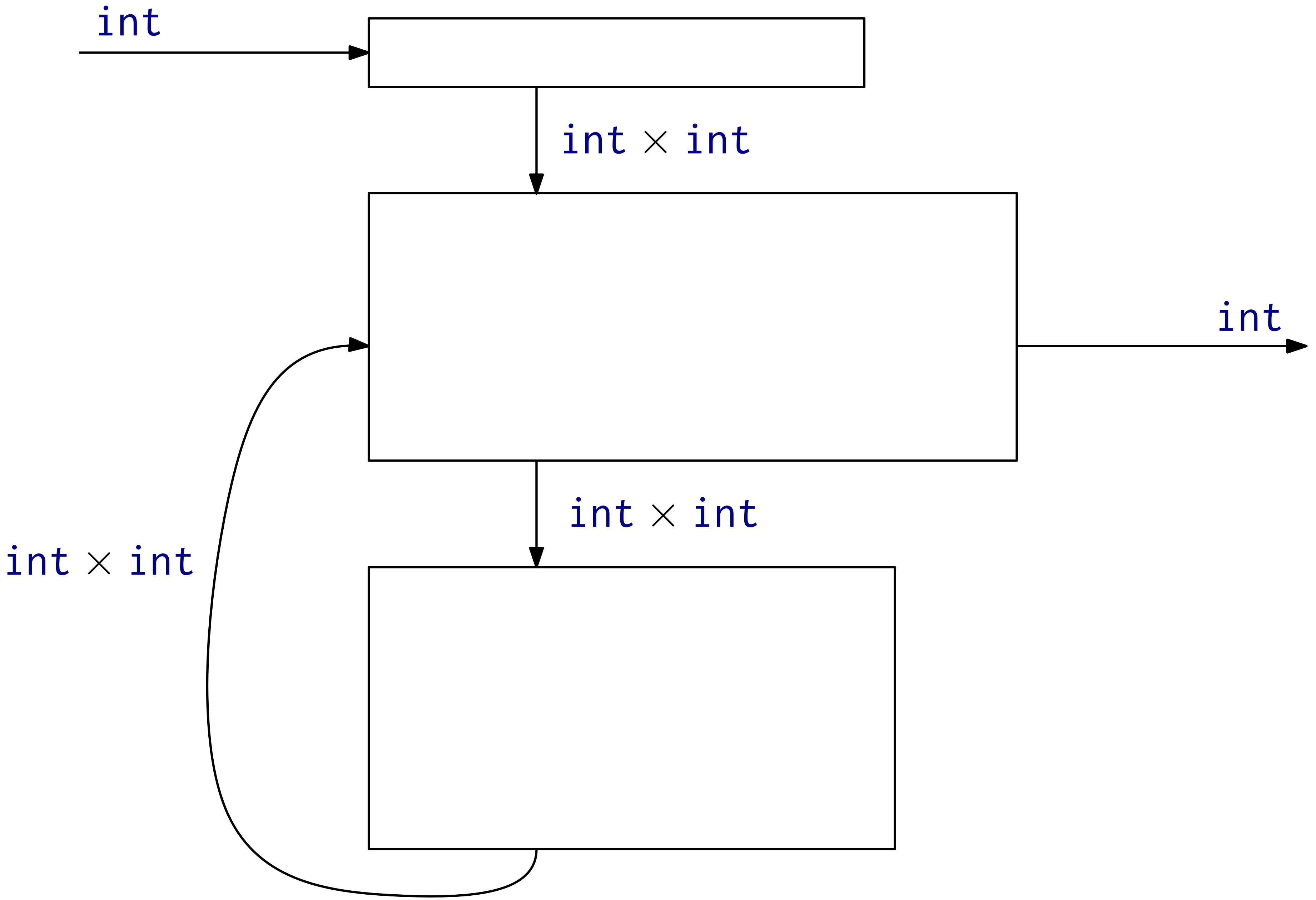
Blocks

$$
\begin{aligned}
&\mathtt{fac}(x : \mathtt{int}) = \mathtt{loop}(x, 1)\\[1em]
&\mathtt{loop}(x : \mathtt{int} \times \mathtt{int}) =\\
&\quad \mathsf{let}\ (n, acc) = x\ \mathsf{in}\\
&\quad \mathsf{let}\ b = eq(n, 1)\ \mathsf{in}\\
&\quad \mathsf{case}\ b\ \mathsf{of}\ \mathtt{inl}(\_) \Rightarrow \mathtt{ret}(acc)\\
&\qquad\qquad\quad ;\ \mathtt{inr}(\_) \Rightarrow \mathtt{body}(n, acc)\\[1em]
&\mathtt{body}(p : \mathtt{int} \times \mathtt{int}) =\\
&\quad \mathsf{let}\ (n, acc) = p\ \mathsf{in}\\
&\quad \mathsf{let}\ n' = sub(n, 1)\ \mathsf{in}\\
&\quad \mathsf{let}\ acc' = mul(acc, n)\ \mathsf{in}\\
&\quad \mathtt{loop}(n', acc')
\end{aligned}
$$

Control-Flow Graphs



Control-Flow Graphs



Operational Semantics

The execution of programs is a series of jumps:

- To begin execution, one jumps with some argument $v : A$ to some block:

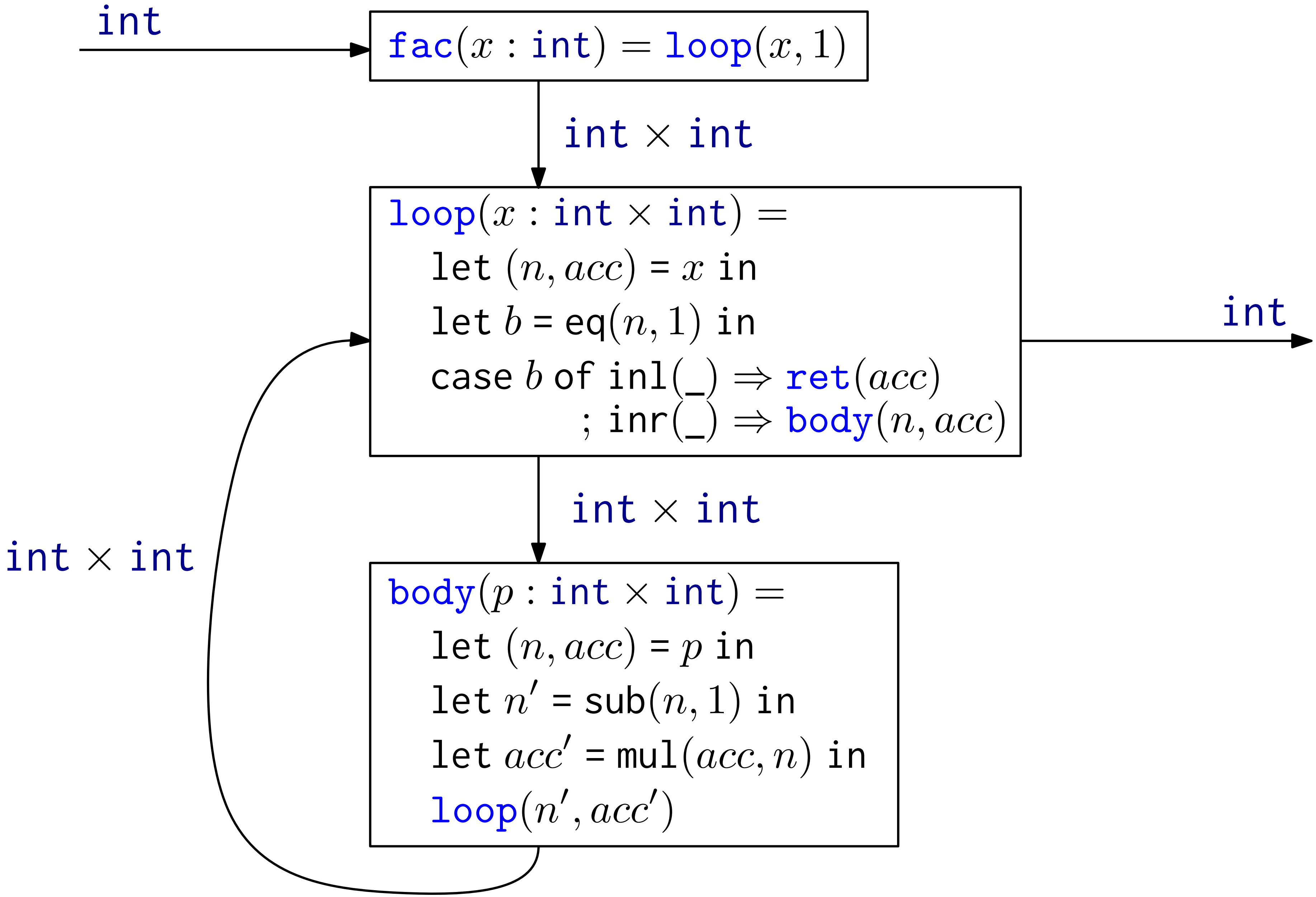
$$\texttt{label}(x : A) = body$$

- This will cause the $body$ to be evaluated.
- Evaluating $body$ ends with a jump to some other block.

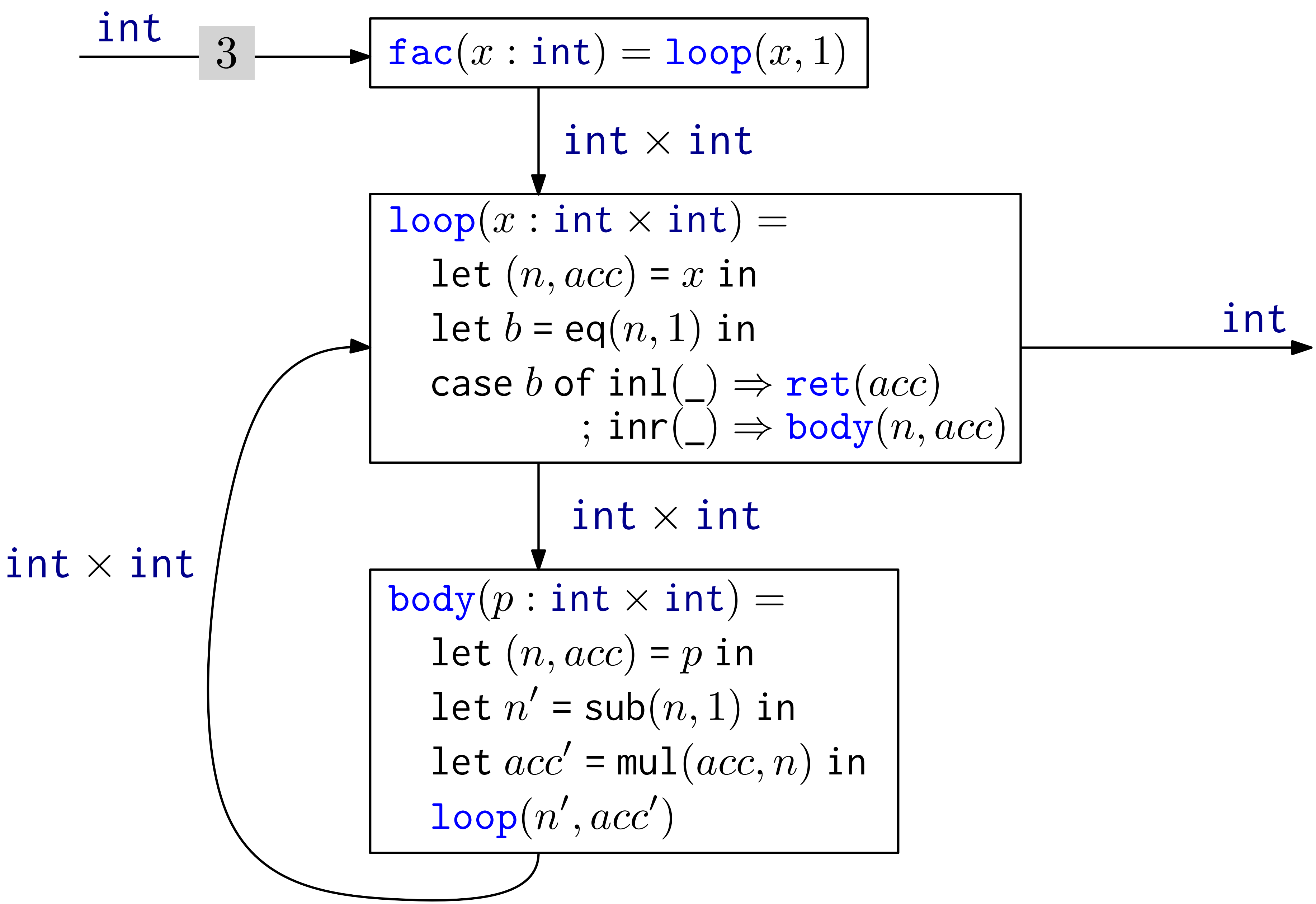
Note

- There is no need for a call stack (or other side-effects).
- Effectful primitive operations may be added, if desired.

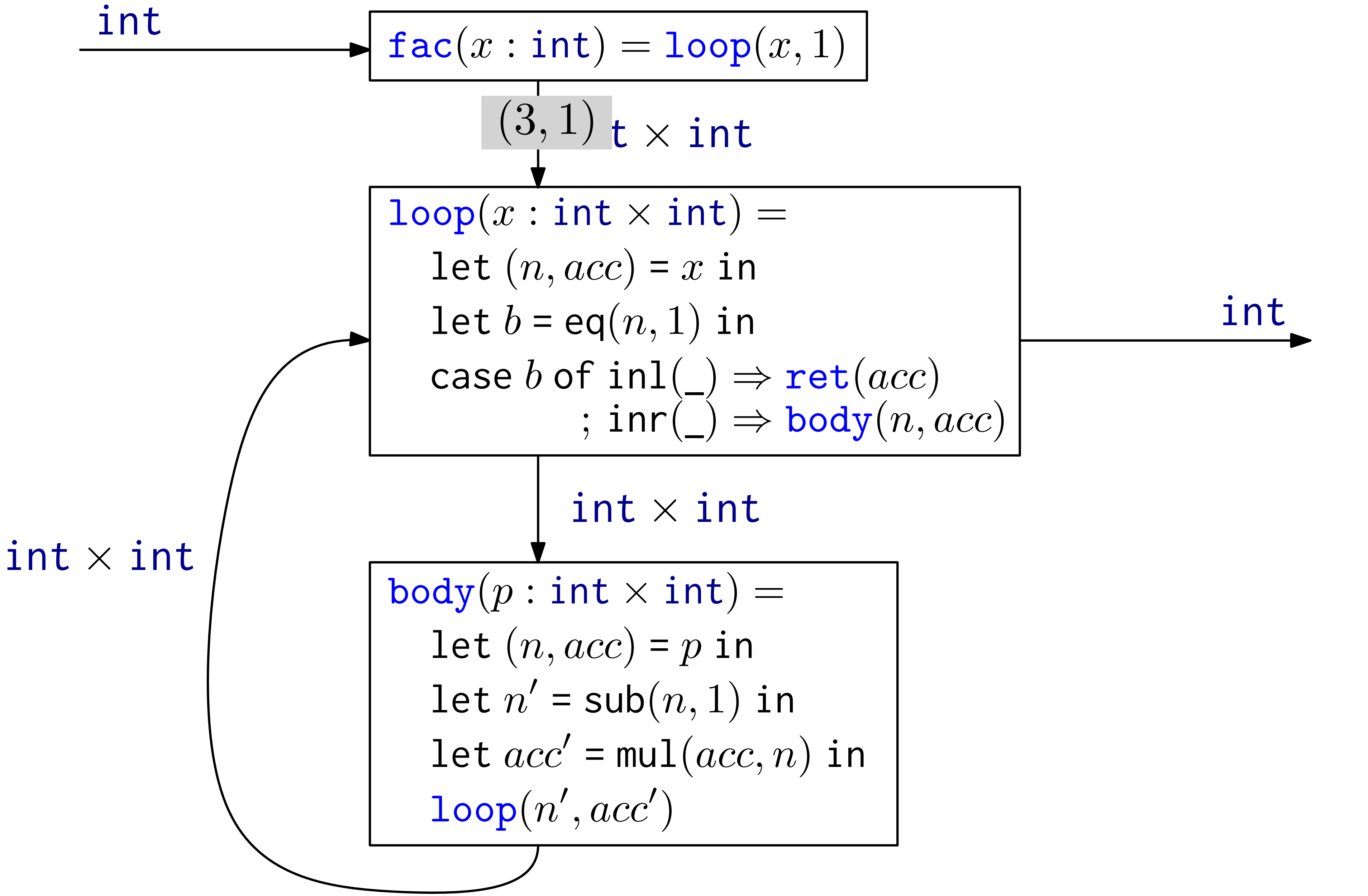
Operational Semantics



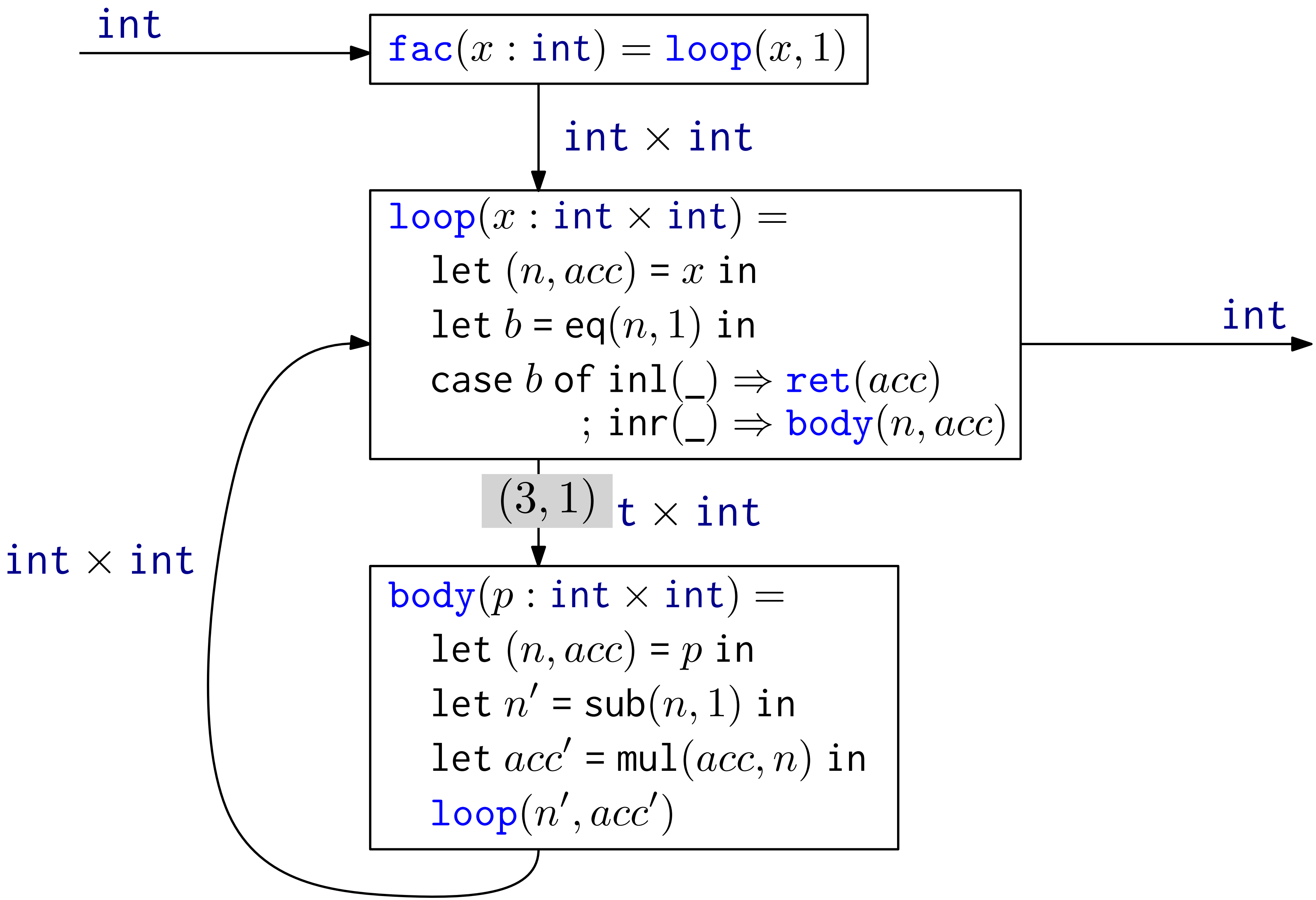
Operational Semantics



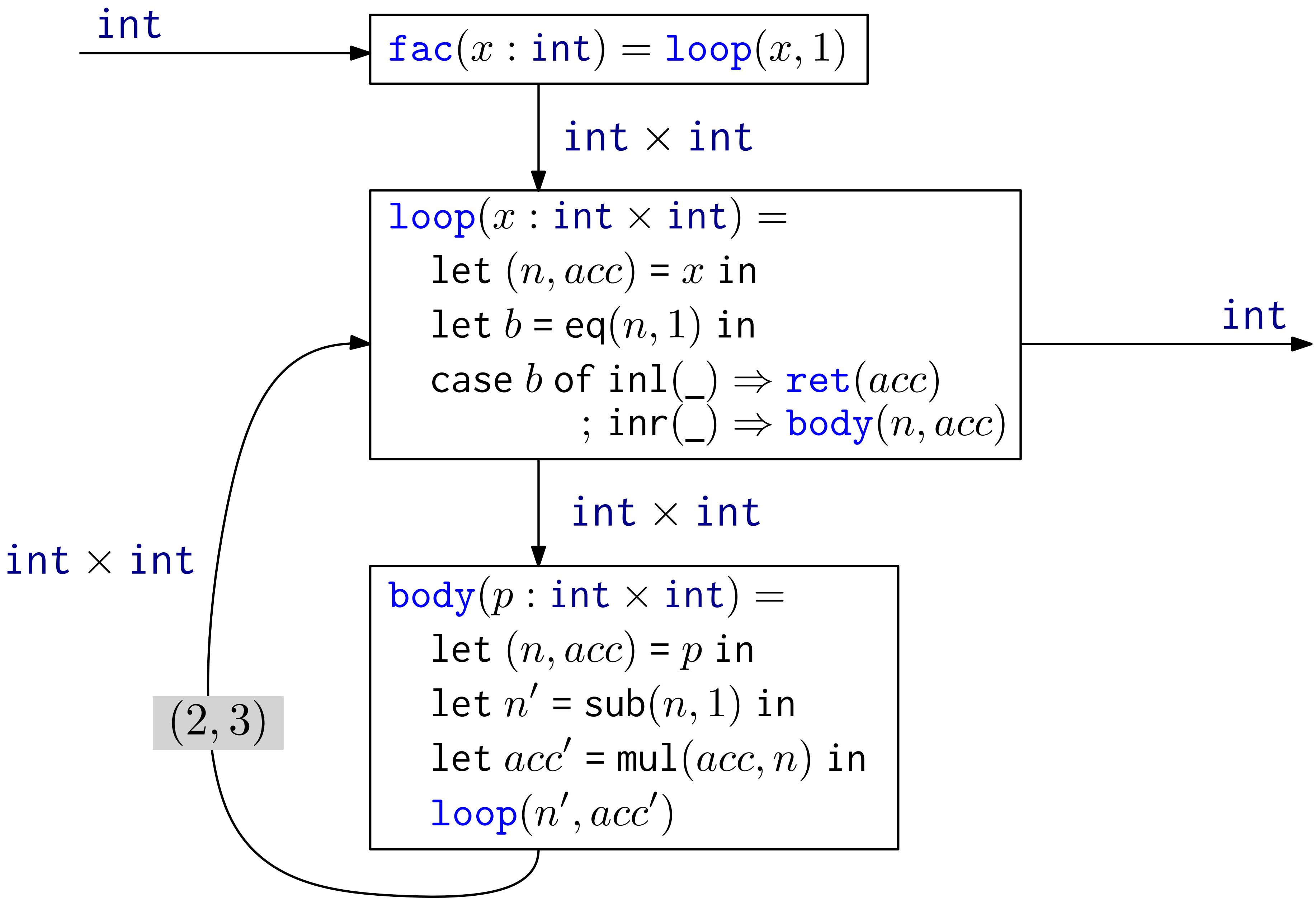
Operational Semantics



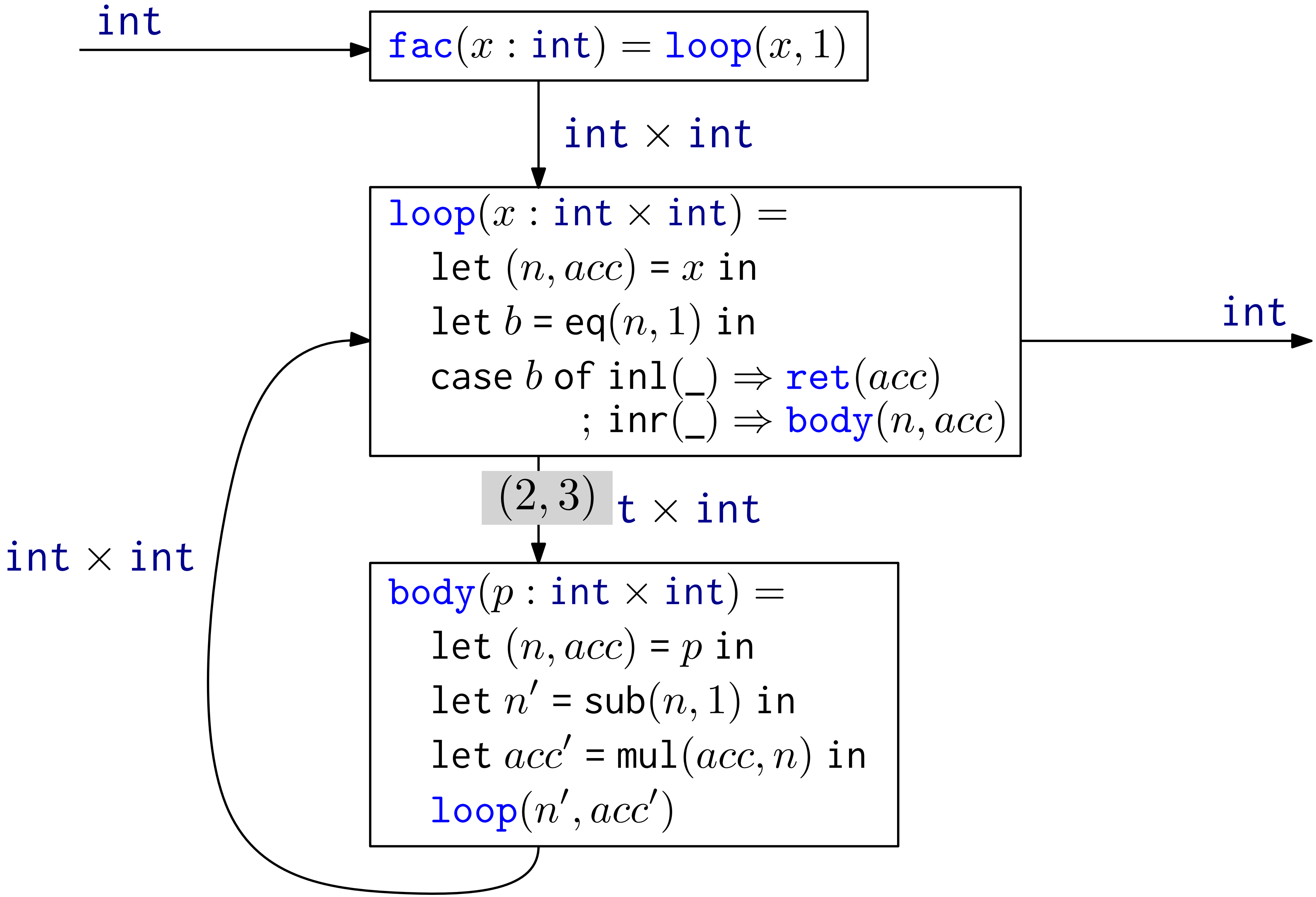
Operational Semantics



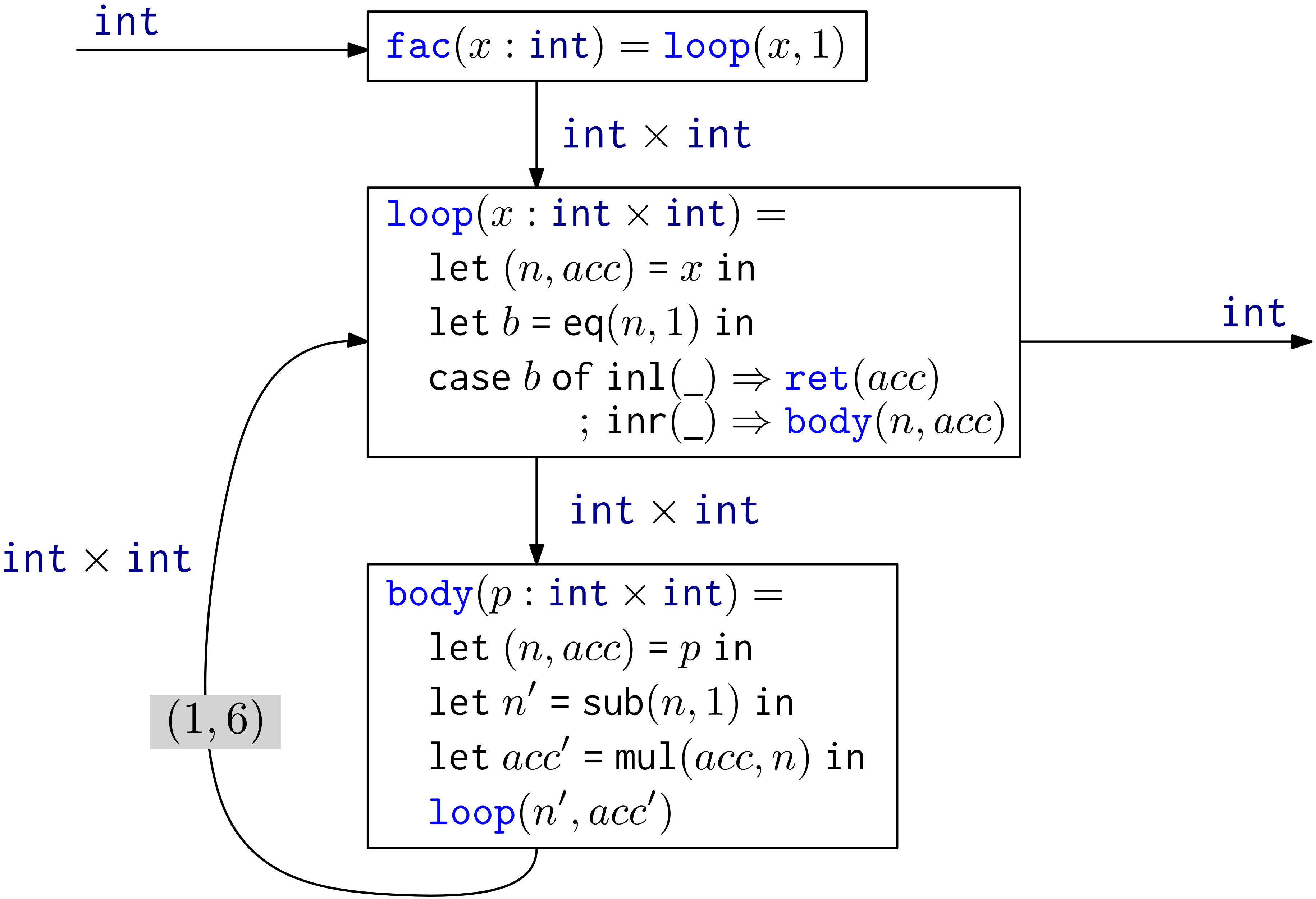
Operational Semantics



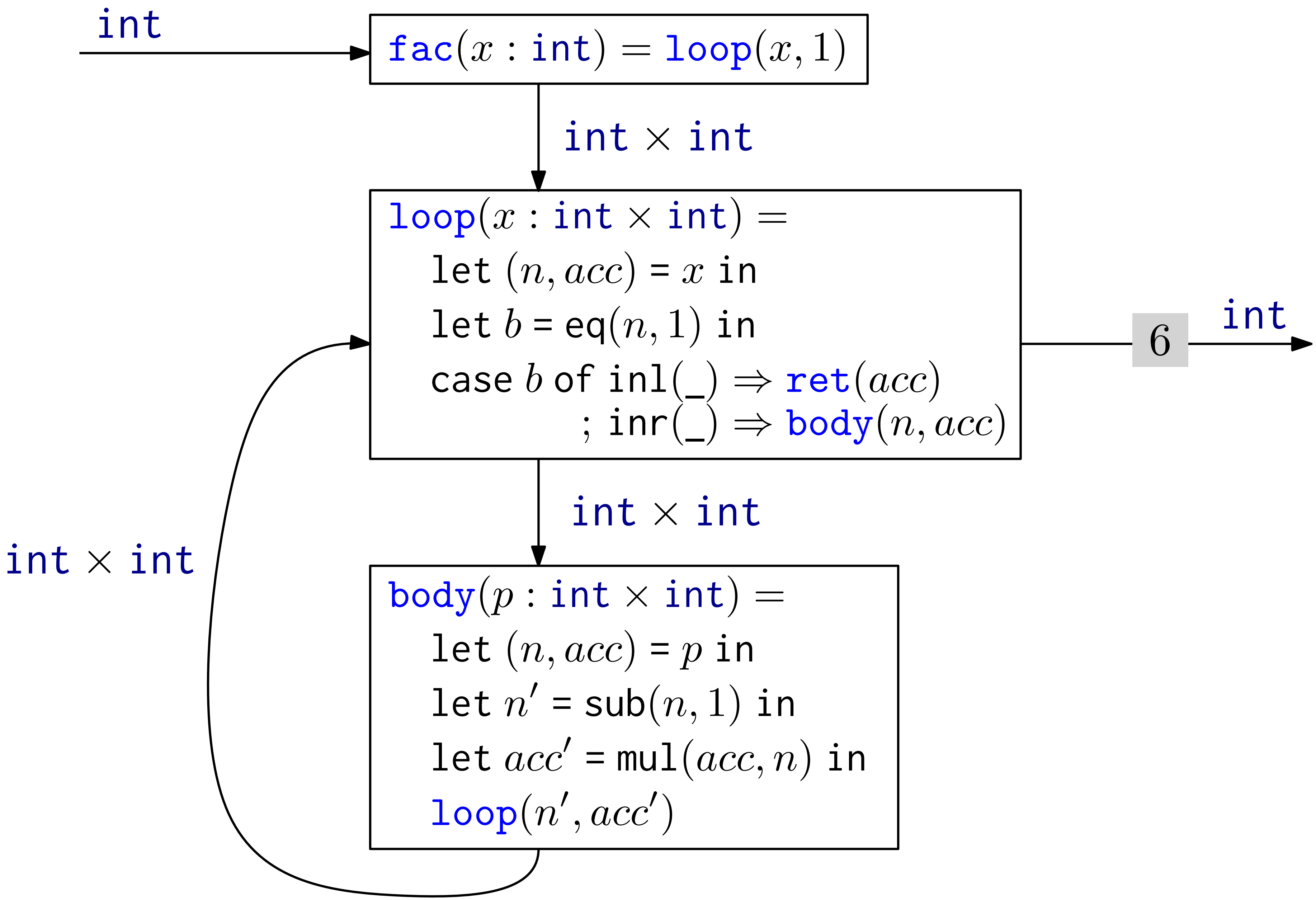
Operational Semantics



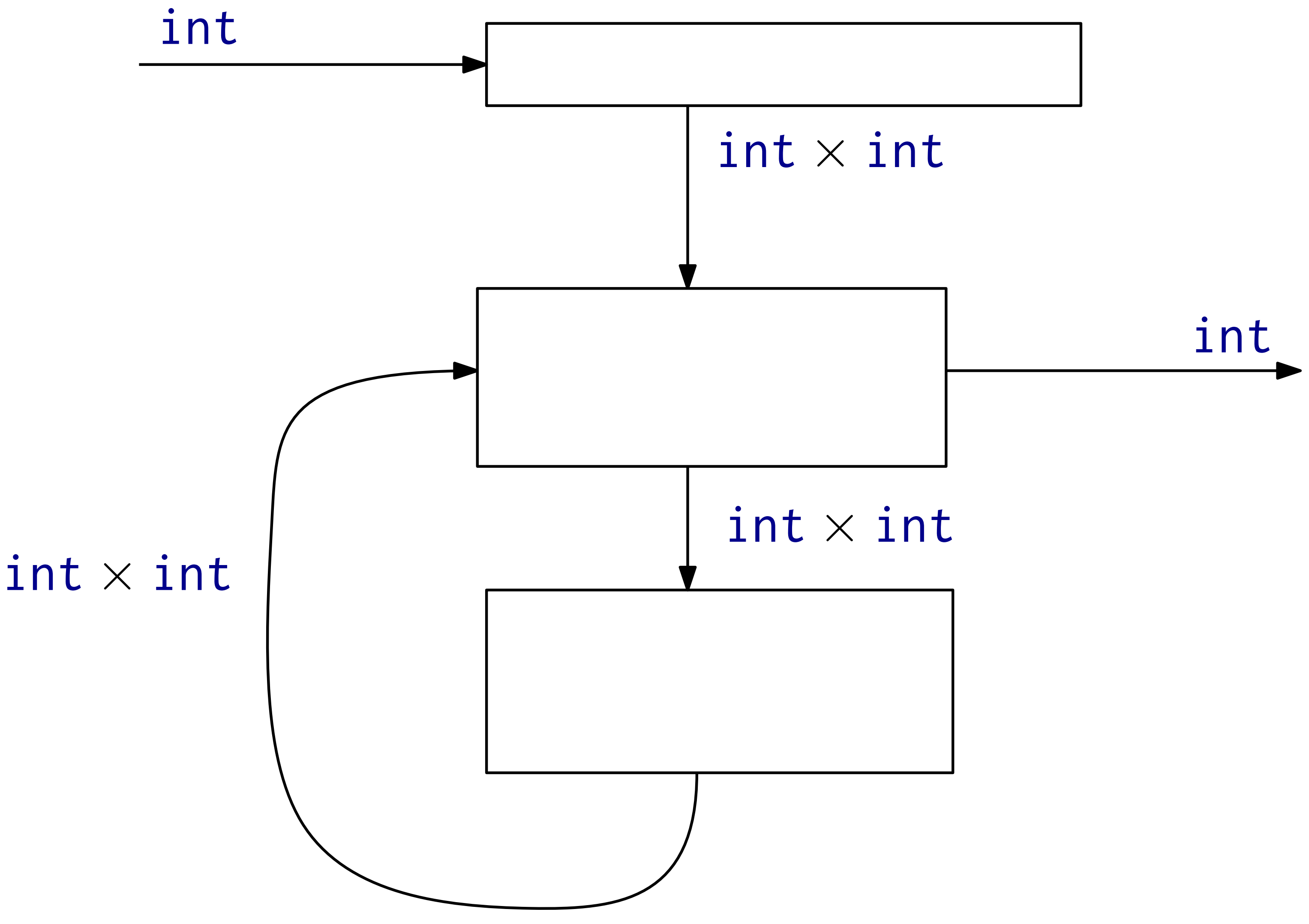
Operational Semantics



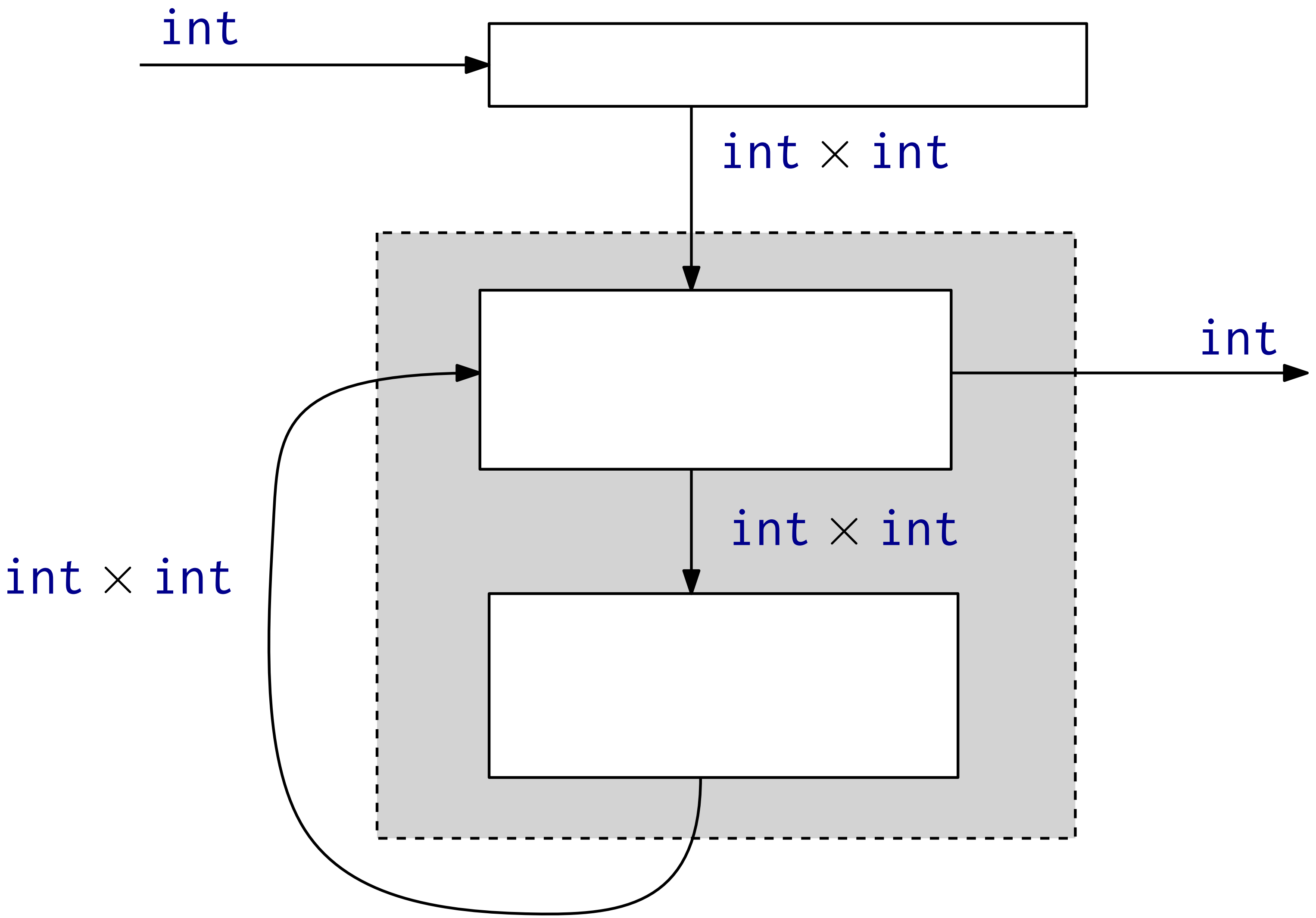
Operational Semantics



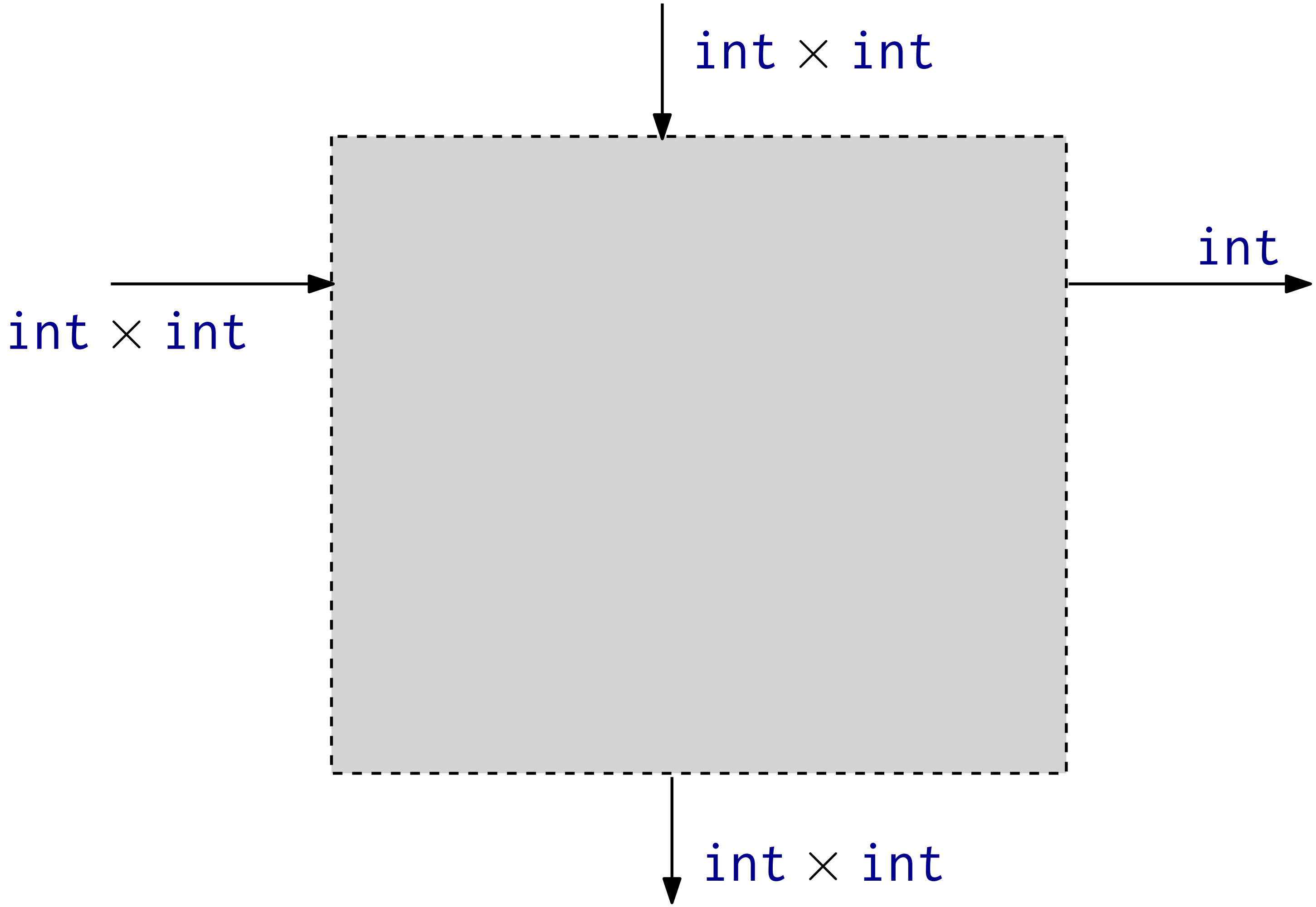
Program Fragment



Program Fragment



Program Fragment

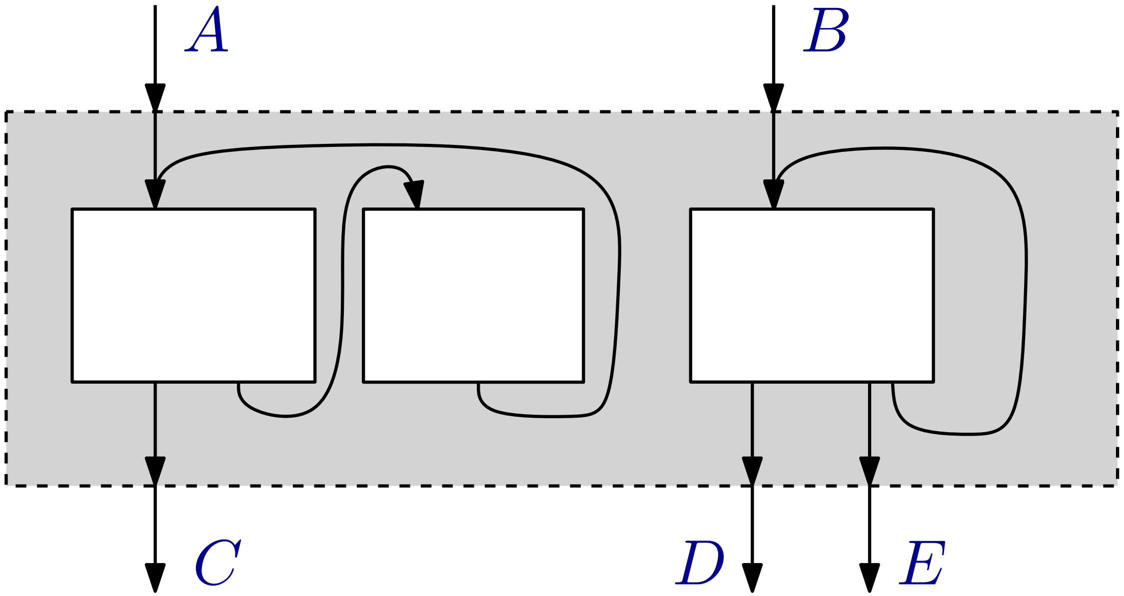


Program Fragment

A **program fragment** is a set of blocks (with pairwise distinct labels) together with

- a list of entry labels (each must be defined in a block),
- a list of exit labels (must not be defined in a block).

Graphical Notation



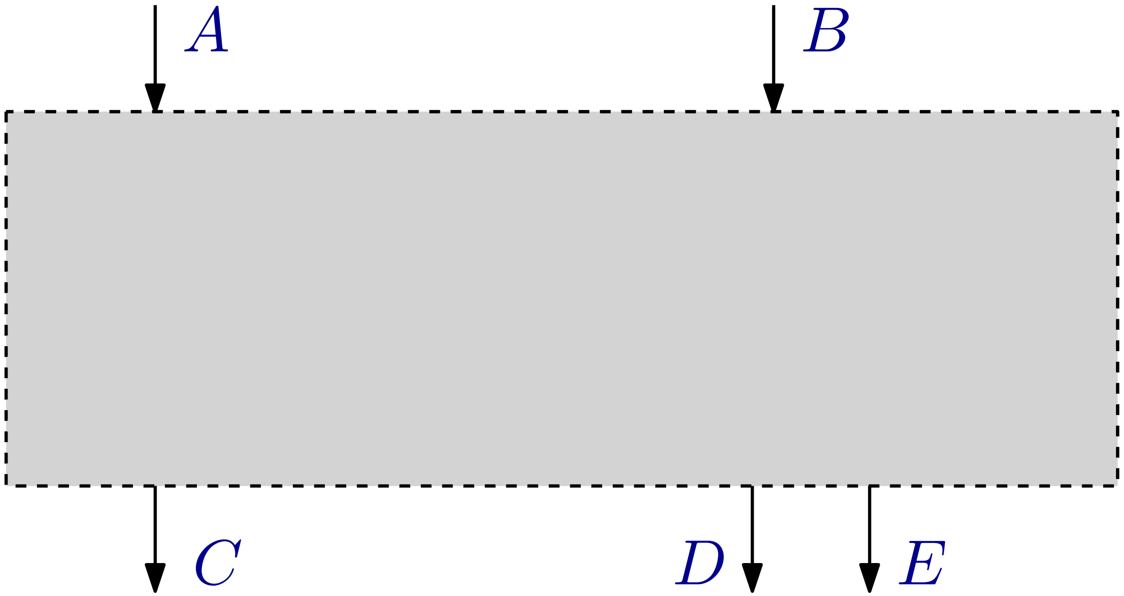
(i.e. control-flow graph with fixed entry- and exit-edges)

Program Fragment

A **program fragment** is a set of blocks (with pairwise distinct labels) together with

- a list of entry labels (each must be defined in a block),
- a list of exit labels (must not be defined in a block).

Graphical Notation

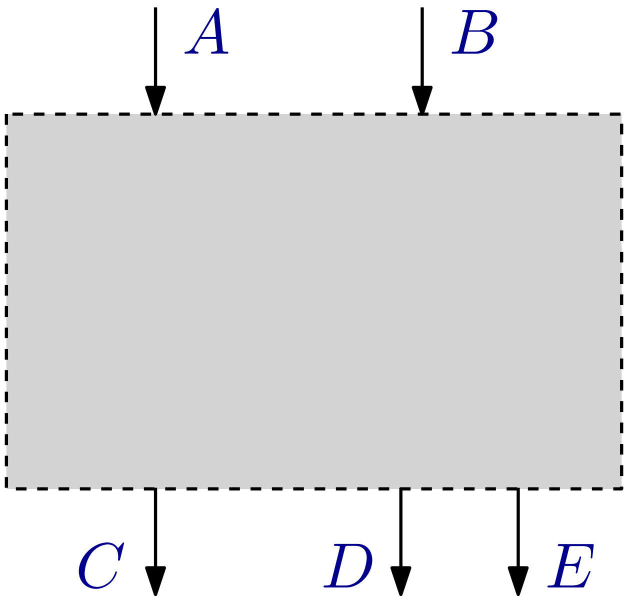


(i.e. control-flow graph with fixed entry- and exit-edges)

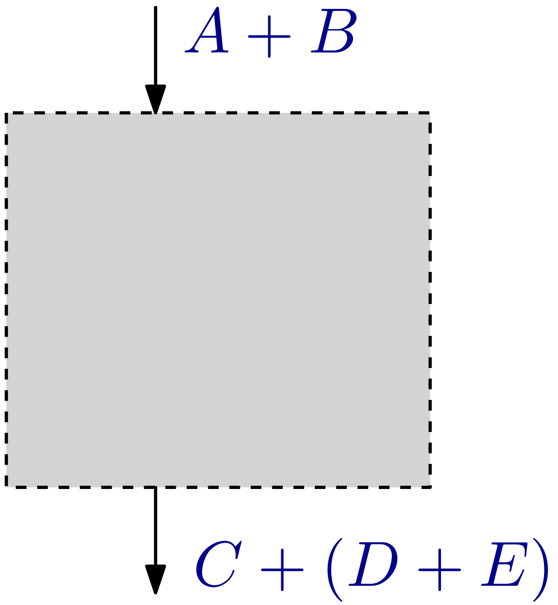
Graphical Notation

In the graphical notation, we omit trivial blocks (e.g. for associativity) and use types to disambiguate.

Example: implicit conversion



$\Longleftrightarrow$



Fragments with Free Variables

We will work with fragments that contain free value variables.

Example program with a free variable $z{:}\mathtt{int}$.

$$\mathtt{sum}(x : \mathtt{int}) = \mathtt{loop}(x, 1)$$

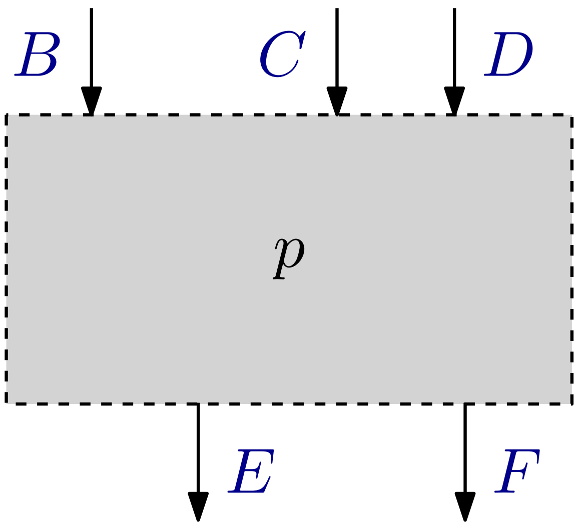
$$\begin{aligned}
&\mathtt{loop}(x : \mathtt{int} \times \mathtt{int}) = \\
&\quad \mathtt{let}\ (n, acc) = x\ \mathtt{in} \\
&\quad \mathtt{let}\ b = eq(n, 1)\ \mathtt{in} \\
&\quad \mathsf{case}\ b\ \mathsf{of}\ \mathtt{inl}(_) \Rightarrow \mathtt{ret}(acc) \\
&\qquad\qquad\quad ;\ \mathtt{inr}(_) \Rightarrow \mathtt{body}(n, acc)
\end{aligned}$$

$$\begin{aligned}
&\mathtt{body}(p : \mathtt{int} \times \mathtt{int}) = \\
&\quad \mathtt{let}\ (n, acc) = p\ \mathtt{in} \\
&\quad \mathtt{let}\ n' = \mathsf{sub}(n, 1)\ \mathtt{in} \\
&\quad \mathtt{let}\ acc' = \mathsf{add}(acc, z)\ \mathtt{in} \\
&\quad \mathtt{loop}(n', acc')
\end{aligned}$$

Graphical Notation — Box

Any program fragment p with a free variable $z{:}A$ can be transformed into a fragment, where z is not free, but is passed around as an argument.

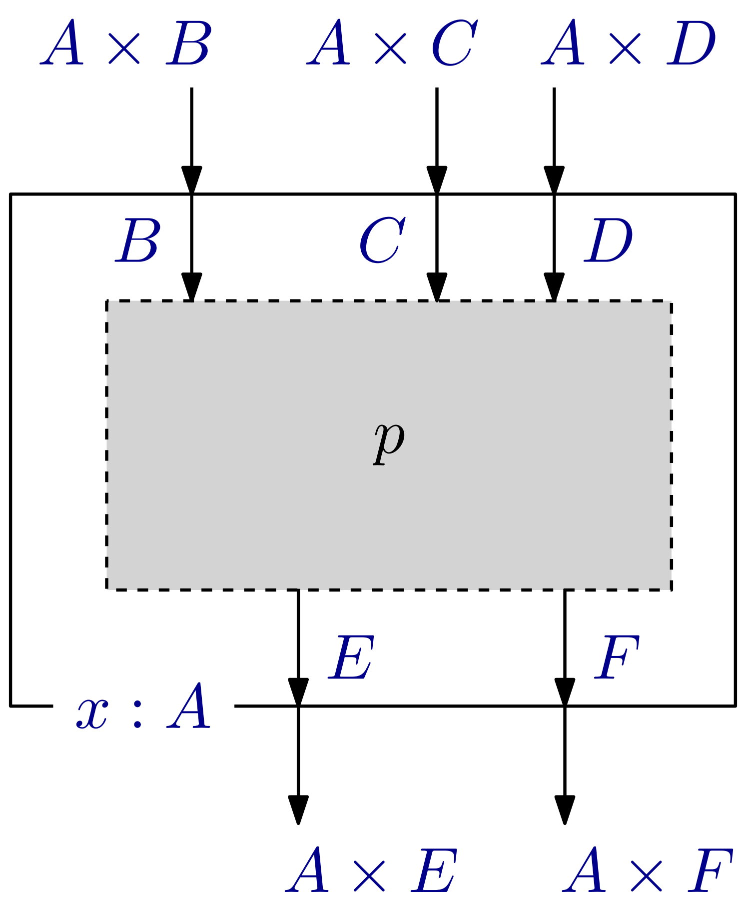
Add a new first parameter of type A to all the blocks in p.



Graphical Notation — Box

Any program fragment p with a free variable $z{:}A$ can be transformed into a fragment, where z is not free, but is passed around as an argument.

Add a new first parameter of type A to all the blocks in p.



Graphical Notation — Box

Any program fragment p with a free variable $z{:}A$ can be transformed into a fragment, where z is not free, but is passed around as an argument.

Add a new first parameter of type A to all the blocks in p.

Example:

$$
\begin{array}{l}
\texttt{label1}(y : B) = \\
\quad \texttt{let}\ b\ \texttt{=}\ \texttt{...}\ \texttt{in} \\
\quad \texttt{case}\ b\ \texttt{of}\ \texttt{inl}(\_) \Rightarrow \texttt{label2}(v) \\
\qquad\qquad\quad ;\ \texttt{inr}(\_) \Rightarrow \texttt{label3}(w)
\end{array}
\quad\Longrightarrow\quad
\begin{array}{l}
\texttt{label1}(z : A \times B) = \\
\quad \texttt{let}\ (x, y)\ \texttt{=}\ z\ \texttt{in} \\
\quad \texttt{let}\ b\ \texttt{=}\ \texttt{...}\ \texttt{in} \\
\quad \texttt{case}\ b\ \texttt{of}\ \texttt{inl}(\_) \Rightarrow \texttt{label2}(x, v) \\
\qquad\qquad\quad ;\ \texttt{inr}(\_) \Rightarrow \texttt{label3}(x, w)
\end{array}
$$

Organising Low-Level Programs

Constructions from Interaction Semantics

Int Construction

Free compact closed completion of a traced symmetric monoidal category [Joyal, Street, Verity 1994].

Captures the core of many constructions, in particular:

- Game Semantics [Abramsky, Jagadeesan, Malacaria 2001]
- Geometry of Interaction [Haghverdi, Scott 2004]

Int Construction

Construction of the **integers** from the **natural numbers**.

Represent integers by pairs $(i^-, i^+) \in \mathbb{N} \times \mathbb{N}$.

$$
\begin{aligned}
(x^-, x^+) \leq (y^-, y^+) \quad &\iff \quad x^+ - x^- \leq y^+ - y^- \\
&\iff \quad x^+ + y^- \leq x^- + y^+
\end{aligned}
$$

Int Construction

Construction of the **integers** from the **natural numbers**.

Represent integers by pairs $(i^-, i^+) \in \mathbb{N} \times \mathbb{N}$.

$$(x^-, x^+) \leq (y^-, y^+) \iff x^+ - x^- \leq y^+ - y^-$$
$$\iff x^+ + y^- \leq x^- + y^+$$

Generalise from the **natural numbers** $(\mathbb{N}, +, 0)$ to a **traced symmetric monoidal category** $(\mathbb{C}, \oplus, 0)$.

Objects: pairs (X^-, X^+) of $\mathbb{C}$-objects

Morphisms:

$$(X^-, X^+) \to (Y^-, Y^+) \iff X^+ \oplus Y^- \to X^- \oplus Y^+ \text{ in } \mathbb{C}$$

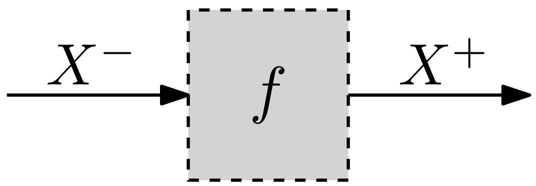
Int Construction for Low-Level Programs

An object (X^-, X^+) models the **interface** of interactive entity.

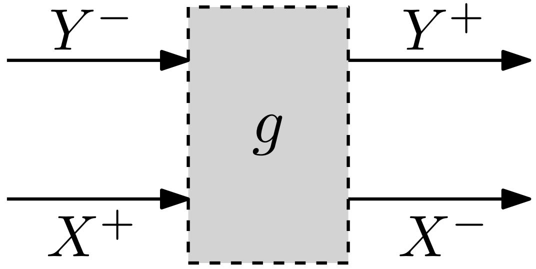
type of possible **answers** from entity

type of possible **questions** to entity

Implementation of interface X:

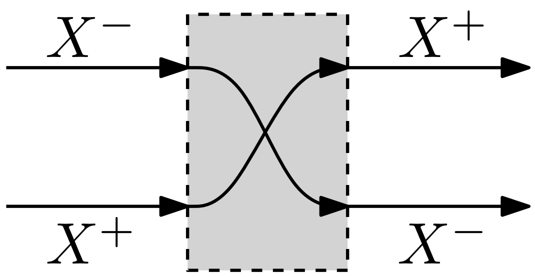


A morphism $X \longrightarrow Y$ is an implementation of interface Y with possible queries to X

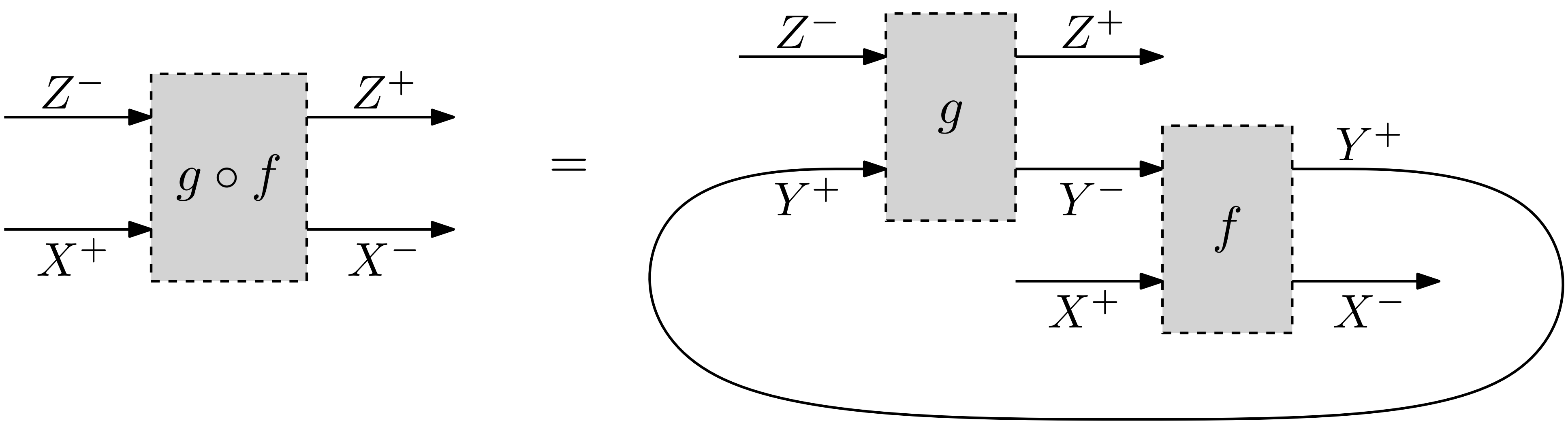


Int Construction for Low-Level Programs

Identity



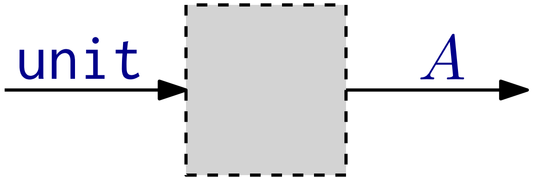
Composition of $f\colon X \longrightarrow Y$ and $g\colon Y \longrightarrow Z$.



Types of Interaction: $[A]$

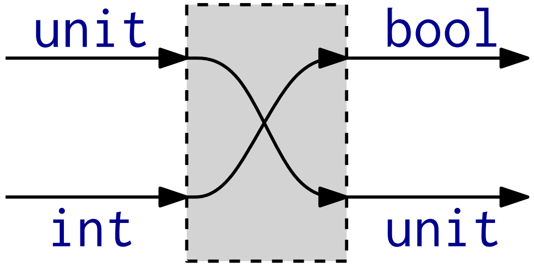
Base $[A]$

- questions: $[A]^- = \texttt{unit}$
- answers: $[A]^+ = A$



Example:

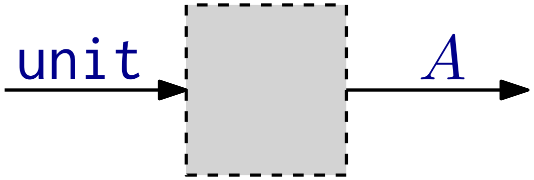
$$\texttt{even}\colon [\texttt{int}] \longrightarrow [\texttt{bool}]$$



Types of Interaction: $[A]$

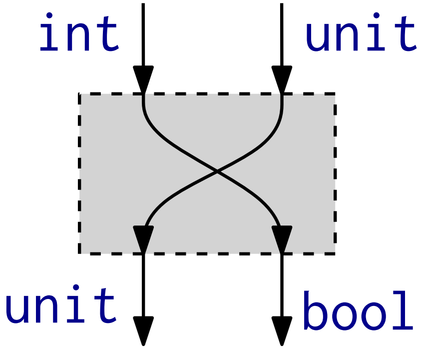
Base $[A]$

- questions: $[A]^- = \texttt{unit}$
- answers: $[A]^+ = A$



Example:

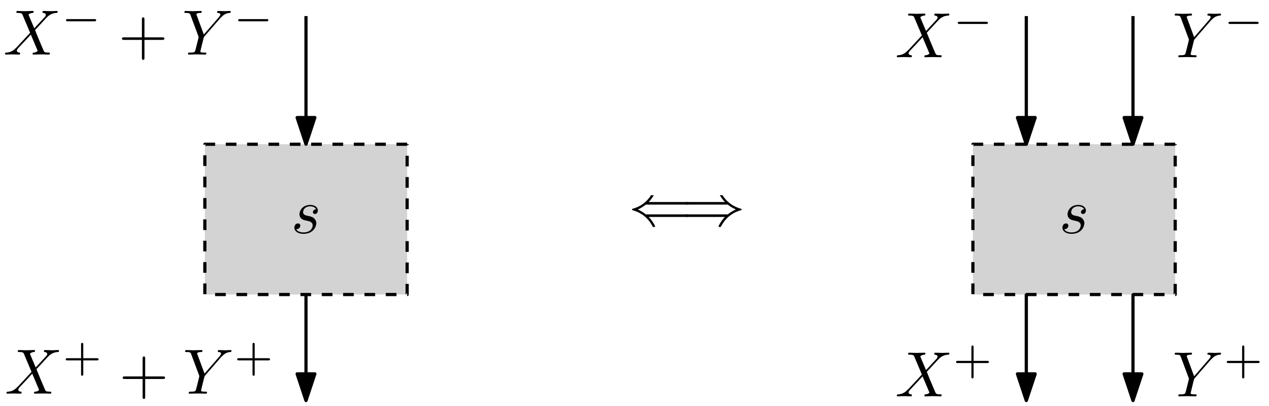
$$\texttt{even}\colon [\texttt{int}] \longrightarrow [\texttt{bool}]$$



Types of Interaction: $X \otimes Y$

Pairs $X \otimes Y$

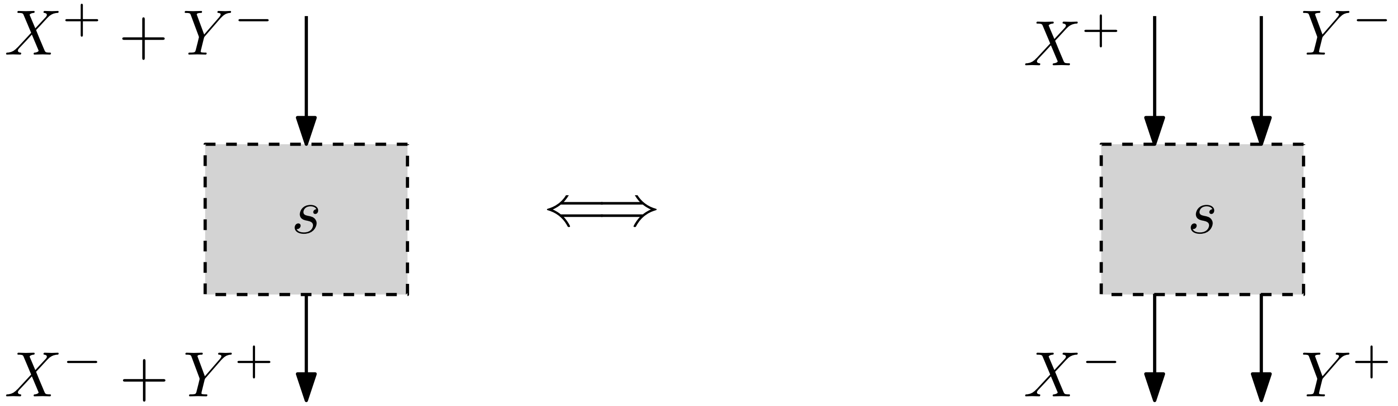
- questions: $(X \otimes Y)^- = X^- + Y^-$
- answers: $(X \otimes Y)^+ = X^+ + Y^+$



Types of Interaction: $X \multimap Y$

Interactive Functions $X \multimap Y$

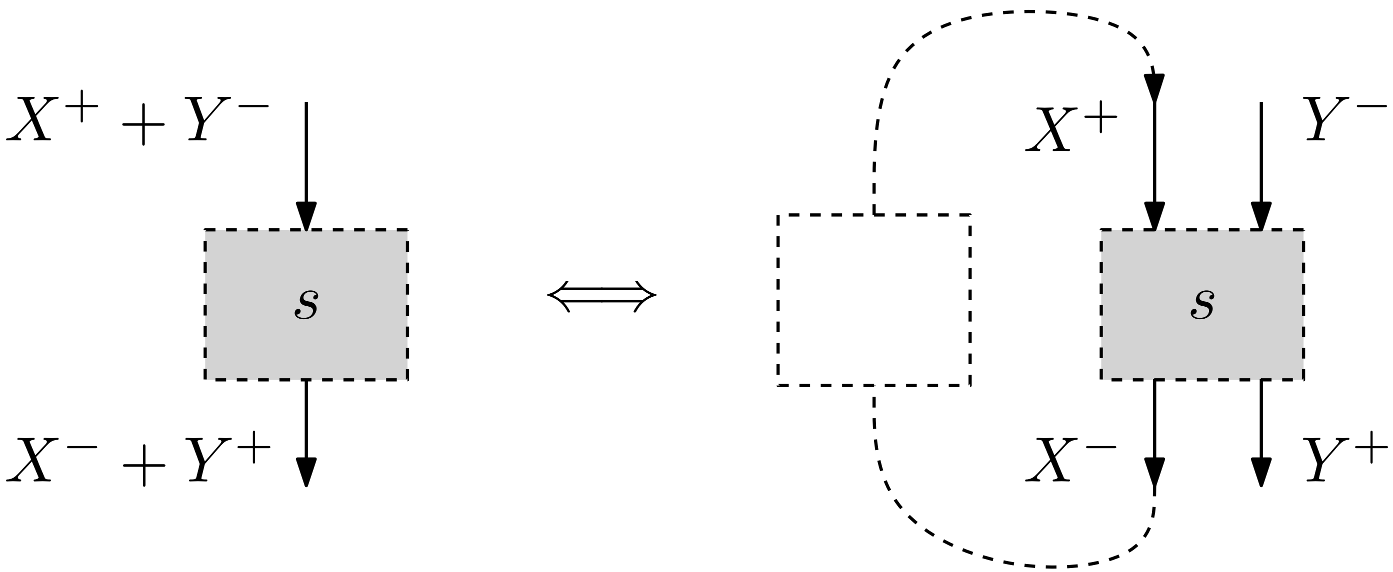
- questions: $(X \multimap Y)^- = X^+ + Y^-$
- answers: $(X \multimap Y)^+ = X^- + Y^+$



Types of Interaction: $X \multimap Y$

Interactive Functions $X \multimap Y$

- questions: $(X \multimap Y)^- = X^+ + Y^-$
- answers: $(X \multimap Y)^+ = X^- + Y^+$



Types of Interaction: $X \multimap Y$

Problem: There is no addition function

$$\mathsf{add}\colon [\mathtt{int}] \multimap [\mathtt{int}] \multimap [\mathtt{int}].$$

We cannot remember values between requests.

Types of Interaction: $A \cdot X$

Subexponential $A \cdot X$

- questions: $(A \cdot X)^- = A \times X^-$
- answers: $(A \cdot X)^+ = A \times X^+$

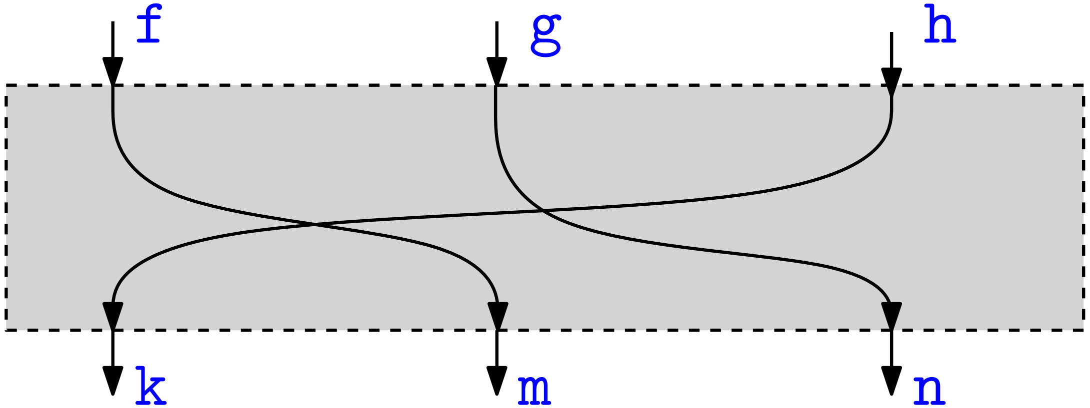
Callee-Save-Invariant

- The value of type A is returned unchanged.
- The answer may not depend on the value of type A.

Types of Interaction: $A \cdot X$

Example

$[A]$	$\multimap$	$A \cdot [B]$	$\multimap$	$[A \times B]$
				$()$
$()$				
a				
		$(a, ())$		
		(a, b)		
				(a, b)



$\mathtt{f}(a : A) = \mathtt{m}(a, ())$

$\mathtt{g}(x : A \times B) = \mathtt{n}(x)$

$\mathtt{h}(x : \mathtt{unit}) = \mathtt{k}(x)$

Types of Interaction: $A \cdot X$

Example

$$\underbrace{(\texttt{unit} + A)}_{S} \cdot ([\texttt{bool}] \multimap [A]) \quad \multimap \quad [A \times A]$$

$()$

$(\texttt{inl}(), ())$

$(\texttt{inl}(), ())$

$(\texttt{inl}(), \texttt{false})$

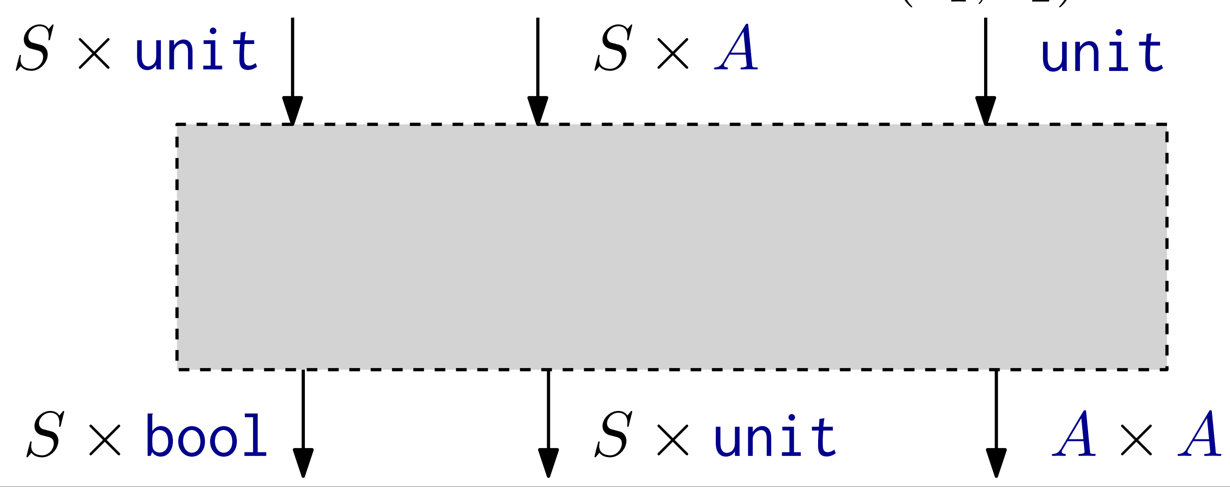
$(\texttt{inl}(), a_1)$

$(\texttt{inr}(a_1), ())$

$(\texttt{inr}(a_1), \texttt{true})$

$(\texttt{inr}(a_1), a_2)$

(a_1, a_2)



Types of Interaction: $A \cdot X$

Examples

$$\texttt{unit} \cdot X \multimap X$$

$$A \cdot (B \cdot X) \multimap (A \times B) \cdot X$$

$$(A + B) \cdot X \multimap (A \cdot X) \otimes (B \cdot X)$$

Types of Interaction: $!X$

Exponential $!X$

$$!X := \mathtt{nat} \cdot X$$

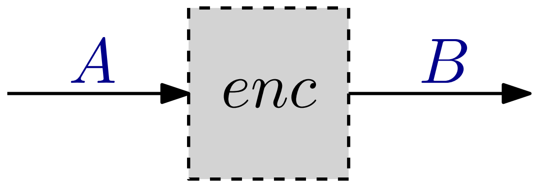
where

$$\begin{aligned} \mathtt{type}\ \mathtt{nat} &= \mathsf{Zero}\ \mathsf{of}\ \mathtt{unit} \\ &\mid \mathsf{Succ}\ \mathsf{of}\ \mathtt{nat} \end{aligned}$$

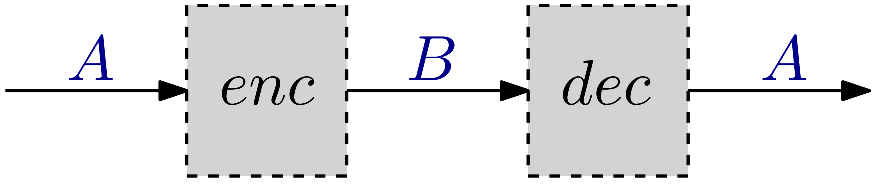
Game Semantics and Geometry of Interaction use this special case.

Retraction $A \lhd B$

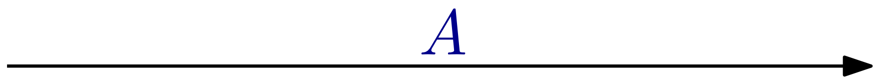
Write $A \lhd B$ if there *exist* two program fragments



such that



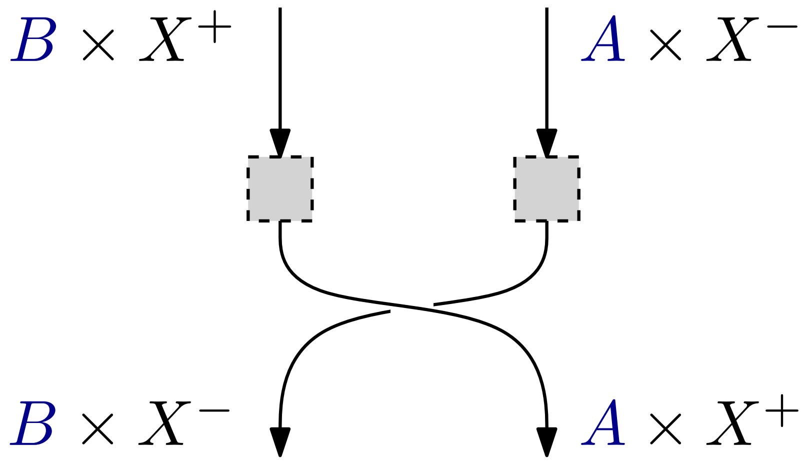
behaves like



Types of Interaction: $!X$

If $A \lhd B$, then we can define a map:

$$B \cdot X \multimap A \cdot X$$



Why not always use $!X$?

- $!X$ requires encoding into `nat`; the compiler for the low-level language would have to undo this for optimisation.
- Encoding details can become visible in some applications, e.g. Bounded Linear Logic [Girard, Scedrov, Scott 1992]

$$(A + B) \cdot X \multimap (A \cdot X) \otimes (B \cdot X)$$

$$!_{p+q} X \multimap !_p X \otimes !_q X$$

$$\begin{aligned} enc(\mathtt{inl}(v)) &= enc(v) \\ enc(\mathtt{inr}(w)) &= p + enc(w) \end{aligned}$$

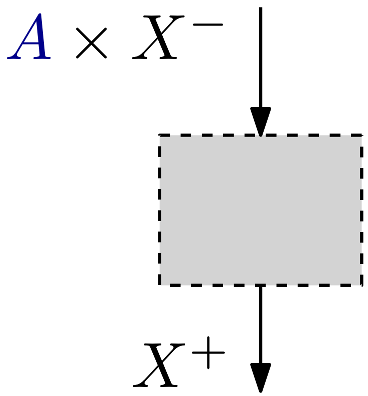
$$!_{2\cdot\max(p,q)} X \multimap !_p X \otimes !_q X$$

$$\begin{aligned} enc(\mathtt{inl}(v)) &= \langle 0, enc(v) \rangle \\ enc(\mathtt{inr}(w)) &= \langle 1, enc(w) \rangle \end{aligned}$$

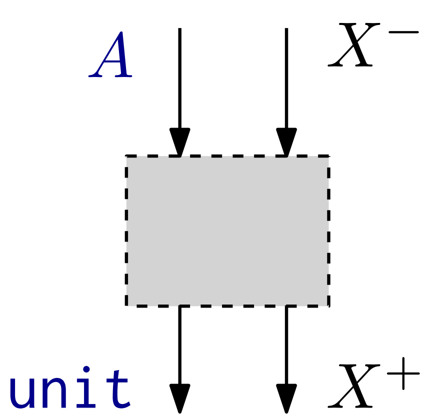
Types of Interaction: $A \to X$

Value Passing $A \to X$

- questions: $(A \to X)^- = A \times X^-$
- answers: $(A \to X)^+ = X^+$



Compare: $[A] \multimap X$



Types of Interaction: $\forall\alpha \lhd A.\, X$ and $\exists\alpha \lhd A.\, X$

Value-Type Polymorphism $\forall\alpha \lhd A.\, X$

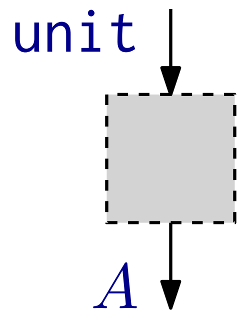
- questions: $(\forall\alpha \lhd A.\, X)^- = (\exists\alpha \lhd A.\, X)^- = X^-[A/\alpha]$
- answers: $(\forall\alpha \lhd A.\, X)^+ = (\exists\alpha \lhd A.\, X)^+ = X^+[A/\alpha]$

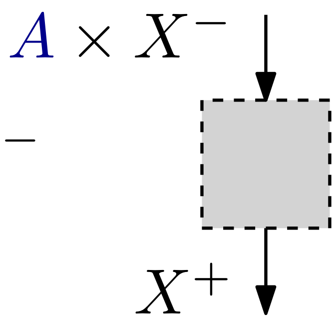
Special cases:

$$\forall\alpha.\, X := \forall\alpha \lhd \mathsf{nat}.\, X$$
$$\exists\alpha.\, X := \forall\alpha \lhd \mathsf{nat}.\, X$$

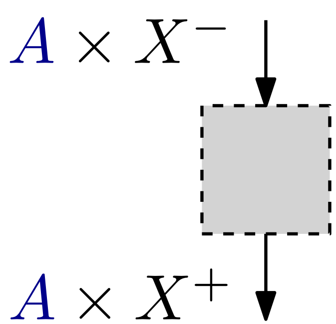
Summary

$$[A]^- = \texttt{unit}$$
$$[A]^+ = A$$



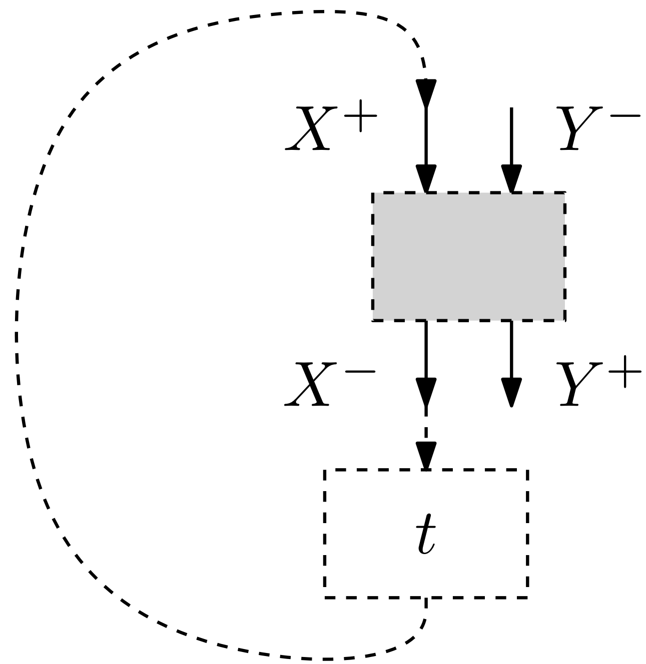


$$(A \to X)^- = A \times X^-$$
$$(A \to X)^+ = X^+$$



$$(A \cdot X)^- = A \times X^-$$
$$(A \cdot X)^+ = A \times X^+$$

$$(X \multimap Y)^- = X^+ + Y^-$$
$$(X \multimap Y)^+ = X^- + Y^+$$



$$(\forall\alpha.X)^- = X^-[\mathsf{nat}/\alpha]$$
$$(\forall\alpha.X)^+ = X^+[\mathsf{nat}/\alpha]$$