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.

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

```assembly
;...
%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]

$$(\bot \land \varphi) \lor \top$$

Opponent

Proponent

*Which of the disjuncts is true?*
Game Semantics for Constructive Logic

[Lorenzen & Lorenz, 1950s]

\[(\bot \land \varphi) \lor \top\]

**Opponent**

*Which of the disjuncts is true?*

**Proponent**

*The left one \( \bot \land \varphi \) is true.*
The left one $\bot \land \varphi$ 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

\[ \text{int} \rightarrow \text{int} \]

Opponent

What does your function return?

Proponent
Game Semantics for Programs

\[ \text{int} \rightarrow \text{int} \]

**Opponent**

*What does your function return?*

**Proponent**

*What is the function argument?*
Game Semantics for Programs

\[ \text{int} \rightarrow \text{int} \]

**Opponent**

*What does your function return?*

**Proponent**

*What is the function argument?*

*The argument is 5.*
Game Semantics for Programs

\[
\text{int} \rightarrow \text{int}
\]

**Opponent**

What does your function return?

The argument is 5.

**Proponent**

What is the function argument?

Then the function returns 6.
Game Semantics for Programs

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

\[ \lambda x. x + 1: \text{int} \rightarrow \text{int} \]

\[ x + 1: \text{int} \]

\[ x: \text{int} \quad 1: \text{int} \]
The strategy of a program derives from strategies of its parts.

What does the function return?

\[ \lambda x. x + 1 : \text{int} \rightarrow \text{int} \]

\[ x + 1 : \text{int} \]

\[ x : \text{int} \]

\[ 1 : \text{int} \]
Game Semantics for Programs

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

\[ \lambda x. x + 1: \text{int} \rightarrow \text{int} \]

What does the function return?

\[ x + 1: \text{int} \]

What is the sum?

\[ x: \text{int} \quad 1: \text{int} \]
The strategy of a program derives from strategies of its parts.

\[
\lambda x. x + 1 : \text{int} \rightarrow \text{int}
\]

What is the sum?

\[
x + 1 : \text{int}
\]

What does the function return?

\[
x : \text{int}
\]

What is value of \(x\)?

\[
1 : \text{int}
\]
The strategy of a program derives from strategies of its parts.

What is value of $x$?

$x + 1: \text{int}$

What is the sum?

What does the function return?

$\lambda x. x + 1: \text{int} \rightarrow \text{int}$
The strategy of a program derives from strategies of its parts.

\[ \lambda x. x + 1 : \text{int} \rightarrow \text{int} \]

What is the function argument?

What does the function return?

What is the sum?

What is value of \( x \)?

What is value of \( x \)?

\[ x + 1 : \text{int} \]

\[ x : \text{int} \]

\[ 1 : \text{int} \]
Game Semantics for Programs

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

\[ \lambda x. x + 1: \text{int} \to \text{int} \]

\[ x + 1: \text{int} \]

\[ x: \text{int} \quad 1: \text{int} \]

The argument is 5.
Game Semantics for Programs

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

\[ \lambda x. x + 1 : \text{int} \rightarrow \text{int} \]

The argument is 5.

The value is 5.

\[ x + 1 : \text{int} \]

\[ x : \text{int} \]

\[ 1 : \text{int} \]
The strategy of a program derives from strategies of its parts.

\[ \lambda x. x + 1: \text{int} \rightarrow \text{int} \]

The argument is 5.

The value is 5.

The value is 5.

The value is 5.
Game Semantics for Programs

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

\[ \lambda x. x + 1 : \text{int} \to \text{int} \]

- The value of 1 is 1.
- The value of \( x + 1 \) is 5.
- The value of \( x \) is 5.

What is the value of 1?
The strategy of a program derives from strategies of its parts.

\[ \lambda x. x + 1: \text{int} \rightarrow \text{int} \]

The argument is 5.

\[ x + 1: \text{int} \]

The value is 5.

\[ x \]

What is value of 1?

\[ 1: \text{int} \]

The value is 1.

The sum is 6.
The strategy of a program derives from strategies of its parts.

The function returns 6.

\[ \lambda x. x + 1 : \text{int} \rightarrow \text{int} \]

The argument is 5.

\[ x + 1 : \text{int} \]

The value is 5.

\[ x \]

What is value of 1?

\[ 1 : \text{int} \]

The sum is 6.

The value is 5.

The value is 1.
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, Jagadeesam & 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.

\( \text{int} \rightarrow \text{int} \)
Implement programs by implementing their interaction strategies.

\[ \text{int} \rightarrow \text{int} \]

Return value?  
Argument is ...  

Return value is ...  
Argument?
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]
- $\pi$-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

\[ A, B ::= \alpha | \text{int} | \text{unit} | A \times B | 0 | A + B | \mu\alpha.A \]

Values

\[ v, w ::= () | n | (v, w) | \text{inl}(v) | \text{inr}(v) | \text{fold}(v) \]

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

Example: Write

\[
\text{type list}\langle\alpha\rangle = \text{Nil of unit} \\
\quad \quad \quad \quad | \text{Cons of } \alpha \times \text{list}\langle\alpha\rangle
\]

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

Programs are constructed from blocks. A **block** has the form

\[
\text{label}(x : A) = \text{body}
\]

where

\[
\text{body} ::= \text{let } x = \text{primop}(v) \text{ in body} \\
| \text{let } (x, y) = v \text{ in body} \\
| \text{let fold}(x) = v \text{ in body} \\
| \text{label}(v) \\
| \text{case } v \text{ of inl}(x) \rightarrow \text{label}_1(v_1); \text{ inr}(y) \rightarrow \text{label}_2(v_2)
\]

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

\[
\begin{align*}
fac(x : \text{int}) &= \text{loop}(x, 1) \\
\text{loop}(x : \text{int} \times \text{int}) &= \\
& \text{let } (n, acc) = x \text{ in} \\
& \text{let } b = eq(n, 1) \text{ in} \\
& \text{case } b \text{ of inl(\_)} \Rightarrow \text{ret}(acc) \\
& \quad \text{; inr(\_)} \Rightarrow \text{body}(n, acc) \\
\text{body}(p : \text{int} \times \text{int}) &= \\
& \text{let } (n, acc) = p \text{ in} \\
& \text{let } n' = \text{sub}(n, 1) \text{ in} \\
& \text{let } acc' = \text{mul}(acc, n) \text{ in} \\
& \text{loop}(n', acc')
\end{align*}
\]
Control-Flow Graphs

\[ \text{fac}(x : \text{int}) = \text{loop}(x, 1) \]

\[ \text{loop}(x : \text{int} \times \text{int}) = \]
\[ \text{let } (n, acc) = x \text{ in} \]
\[ \text{let } b = \text{eq}(n, 1) \text{ in} \]
\[ \text{case } b \text{ of } \text{inl(}_) \Rightarrow \text{ret}(acc) \]
\[ ; \text{inr(}_) \Rightarrow \text{body}(n, acc) \]

\[ \text{body}(p : \text{int} \times \text{int}) = \]
\[ \text{let } (n, acc) = p \text{ in} \]
\[ \text{let } n' = \text{sub}(n, 1) \text{ in} \]
\[ \text{let } acc' = \text{mul}(acc, n) \text{ in} \]
\[ \text{loop}(n', acc') \]
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:
  $$\text{label}(x : A) = \text{body}$$
- This will cause the $\text{body}$ to be evaluated.
- Evaluating $\text{body}$ 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**

\[ \text{fac}(x : \text{int}) = \text{loop}(x, 1) \]

\[ \text{loop}(x : \text{int} \times \text{int}) = \]
\[ \text{let } (n, \text{acc}) = x \text{ in} \]
\[ \text{let } b = \text{eq}(n, 1) \text{ in} \]
\[ \text{case } b \text{ of } \text{inl}(\_ ) \Rightarrow \text{ret}(\text{acc}) \]
\[ ; \text{inr}(\_ ) \Rightarrow \text{body}(n, \text{acc}) \]

\[ \text{body}(p : \text{int} \times \text{int}) = \]
\[ \text{let } (n, \text{acc}) = p \text{ in} \]
\[ \text{let } n' = \text{sub}(n, 1) \text{ in} \]
\[ \text{let } \text{acc}' = \text{mul}(\text{acc}, n) \text{ in} \]
\[ \text{loop}(n', \text{acc}') \]
Operational Semantics

fac(x : int) = loop(x, 1)

loop(x : int × int) =
let (n, acc) = x in
let b = eq(n, 1) in
case b of inl(?) ⇒ ret(acc)
; inr(?) ⇒ body(n, acc)

body(p : int × int) =
let (n, acc) = p in
let n' = sub(n, 1) in
let acc' = mul(acc, n) in
loop(n', acc')
**Operational Semantics**

\[ \text{fac}(x : \text{int}) = \text{loop}(x, 1) \]

\[ (3, 1) \]

\[ (3, 1) \times \text{int} \]

\[ \text{loop}(x : \text{int} \times \text{int}) = \]
\[
\begin{align*}
\text{let } (n, acc) &= x \text{ in} \\
\text{let } b &= \text{eq}(n, 1) \text{ in} \\
\text{case } b \text{ of} \text{inl}(?) &\Rightarrow \text{ret}(acc) \\
; \text{inr}(?) &\Rightarrow \text{body}(n, acc)
\end{align*}
\]

\[ \text{int} \]

\[ \text{int} \times \text{int} \]

\[ \text{int} \times \text{int} \]

\[ \text{body}(p : \text{int} \times \text{int}) = \]
\[
\begin{align*}
\text{let } (n, acc) &= p \text{ in} \\
\text{let } n' &= \text{sub}(n, 1) \text{ in} \\
\text{let } acc' &= \text{mul}(acc, n) \text{ in} \\
\text{loop}(n', acc')
\end{align*}
\]
Operational Semantics

\[
\text{fac}(x : \text{int}) = \text{loop}(x, 1)
\]

\[
\text{loop}(x : \text{int} \times \text{int}) =
\begin{align*}
&\text{let} \ (n, acc) = x \ \text{in} \\
&\text{let} \ b = \text{eq}(n, 1) \ \text{in} \\
&\text{case} \ b \ \text{of} \ \text{inl}(\_ ) \Rightarrow \text{ret}(acc) \\
&\quad; \ \text{inr}(\_ ) \Rightarrow \text{body}(n, acc)
\end{align*}
\]

\[
\text{body}(p : \text{int} \times \text{int}) =
\begin{align*}
&\text{let} \ (n, acc) = p \ \text{in} \\
&\text{let} \ n' = \text{sub}(n, 1) \ \text{in} \\
&\text{let} \ acc' = \text{mul}(acc, n) \ \text{in} \\
&\text{loop}(n', acc')
\end{align*}
\]
Operational Semantics

\[ \text{fac}(x : \text{int}) = \text{loop}(x, 1) \]

\[ \text{loop}(x : \text{int} \times \text{int}) = \]
\[ \begin{array}{l}
\text{let } (n, \text{acc}) = x \text{ in} \\
\text{let } b = \text{eq}(n, 1) \text{ in} \\
\text{case } b \text{ of inl(\_)} \Rightarrow \text{ret}(\text{acc}) \\
; \text{inr(\_)} \Rightarrow \text{body}(n, \text{acc})
\end{array} \]

\[ \text{body}(p : \text{int} \times \text{int}) = \]
\[ \begin{array}{l}
\text{let } (n, \text{acc}) = p \text{ in} \\
\text{let } n' = \text{sub}(n, 1) \text{ in} \\
\text{let } acc' = \text{mul}(acc, n) \text{ in} \\
\text{loop}(n', acc')
\end{array} \]
Operational Semantics

\[ \text{fact}(x: \text{int}) = \text{loop}(x, 1) \]

\[ \text{loop}(x: \text{int} \times \text{int}) = \]
\[ \begin{align*}
&\text{let } (n, acc) = x \text{ in} \\
&\text{let } b = \text{eq}(n, 1) \text{ in} \\
&\text{case } b \text{ of } \text{inl}(?) \Rightarrow \text{ret}(acc) \\
&\quad ; \quad \text{inr}(?) \Rightarrow \text{body}(n, acc)
\end{align*} \]

\[ \text{body}(p: \text{int} \times \text{int}) = \]
\[ \begin{align*}
&\text{let } (n, acc) = p \text{ in} \\
&\text{let } n' = \text{sub}(n, 1) \text{ in} \\
&\text{let } acc' = \text{mul}(acc, n) \text{ in} \\
&\text{loop}(n', acc')
\end{align*} \]
Operational Semantics

\[
\text{fac}(x: \text{int}) = \text{loop}(x, 1)
\]

\[
\text{loop}(x: \text{int} \times \text{int}) =
\begin{align*}
&\text{let } (n, acc) = x \text{ in} \\
&\text{let } b = \text{eq}(n, 1) \text{ in} \\
&\text{case } b \text{ of } \text{inl}(\_ ) \Rightarrow \text{ret}(acc) \\
&; \text{inr}(\_ ) \Rightarrow \text{body}(n, acc)
\end{align*}
\]

\[
\text{body}(p: \text{int} \times \text{int}) =
\begin{align*}
&\text{let } (n, acc) = p \text{ in} \\
&\text{let } n' = \text{sub}(n, 1) \text{ in} \\
&\text{let } acc' = \text{mul}(acc, n) \text{ in} \\
&\text{loop}(n', acc')
\end{align*}
\]
\[ \text{fac}(x : \text{int}) = \text{loop}(x, 1) \]

\[ \text{loop}(x : \text{int} \times \text{int}) = \]
\[
\text{let } (n, acc) = x \text{ in }
\text{let } b = \text{eq}(n, 1) \text{ in }
\text{case } b \text{ of } \text{inl}() \Rightarrow \text{ret}(acc)
\]; \text{inr}() \Rightarrow \text{body}(n, acc)
\]

\[ \text{body}(p : \text{int} \times \text{int}) = \]
\[
\text{let } (n, acc) = p \text{ in }
\text{let } n' = \text{sub}(n, 1) \text{ in }
\text{let } acc' = \text{mul}(acc, n) \text{ in }
\text{loop}(n', acc')
\]
Program Fragment

\[ \text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int} \]
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)
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
Fragments with Free Variables

We will work with fragments that contain free value variables.

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

\[
\begin{align*}
\text{sum}(x : \text{int}) &= \text{loop}(x, 1) \\
\text{loop}(x : \text{int} \times \text{int}) &= \\
&\quad \text{let } (n, acc) = x \text{ in} \\
&\quad \text{let } b = \text{eq}(n, 1) \text{ in} \\
&\quad \text{case } b \text{ of } \text{inl}(\_)) \Rightarrow \text{ret}(acc) \\
&\quad ; \text{inr}(\_)) \Rightarrow \text{body}(n, acc) \\
\text{body}(p : \text{int} \times \text{int}) &= \\
&\quad \text{let } (n, acc) = p \text{ in} \\
&\quad \text{let } n' = \text{sub}(n, 1) \text{ in} \\
&\quad \text{let } acc' = \text{add}(acc, z) \text{ in} \\
&\quad \text{loop}(n', acc')
\end{align*}
\]
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 \( p \).
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 $p$.
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:

\[
\begin{align*}
\text{label1}(y : B) &= \\
&\quad \text{let } b = \ldots \text{ in} \\
&\quad \text{case } b \text{ of } \text{inl}(\_ ) \Rightarrow \text{label2}(v) \\
&\quad \quad \text{; inr}(\_ ) \Rightarrow \text{label3}(w)
\end{align*}
\]

\[
\begin{align*}
\text{label1}(z : A \times B) &= \\
&\quad \text{let } (x, y) = z \text{ in} \\
&\quad \text{let } b = \ldots \text{ in} \\
&\quad \quad \text{case } b \text{ of } \text{inl}(\_ ) \Rightarrow \text{label2}(x, v) \\
&\quad \quad \quad \text{; inr}(\_ ) \Rightarrow \text{label3}(x, w)
\end{align*}
\]
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]
Construction of the **integers** from the **natural numbers**.

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

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

$$\iff x^+ + y^- \leq x^- + y^+$$
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:**

\[(X^-, X^+) \rightarrow (Y^-, Y^+) \iff X^+ \oplus Y^- \rightarrow X^- \oplus Y^+ \text{ in } \mathbb{C}\]
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 \(X\):

\[
\begin{align*}
X^- & \rightarrow f & \rightarrow X^+ \\
Y^- & \rightarrow g & \rightarrow Y^+
\end{align*}
\]

A morphism \(X \rightarrow Y\) is an implementation of interface \(Y\) with possible queries to \(X\).
**Int Construction for Low-Level Programs**

**Identity**

Composition of $f: X \rightarrow Y$ and $g: Y \rightarrow Z$.

$$g \circ f =$$
Types of Interaction: \([A]\)

**Base \([A]\)**
- questions: \([A]^- = \text{unit}\)
- answers: \([A]^+ = A\)

**Example:**

Even: \([\text{int}] \rightarrow [\text{bool}]\)
Types of Interaction: \([A]\)

**Base \([A]\)**
- questions: \([A]^- = \text{unit}\)
- answers: \([A]^+ = A\)

**Example:**

\[
\text{even: } \text{[int]} \rightarrow \text{[bool]}
\]
Types of Interaction: $X \otimes Y$

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

\[
\begin{array}{c}
X^- + Y^- \\
\downarrow \\
\quad s \\
X^+ + Y^+ \\
\end{array}
\quad \iff \quad 
\begin{array}{c}
X^- \\
\downarrow \\
\quad s \\
X^+ \\
\end{array}
\quad \iff 
\begin{array}{c}
X^- \\
\downarrow \\
\quad s \\
X^+ \\
\end{array}
\quad \iff 
\begin{array}{c}
Y^- \\
\downarrow \\
\quad s \\
Y^+ \\
\end{array}
\quad \iff 
\begin{array}{c}
Y^- \\
\downarrow \\
\quad s \\
Y^+ \\
\end{array}
\]
Types of Interaction: $X \rightarrow Y$

Interactive Functions $X \rightarrow Y$

- questions: $(X \rightarrow Y)^- = X^+ + Y^-$
- answers: $(X \rightarrow Y)^+ = X^- + Y^+$
Types of Interaction: $X \rightarrow Y$

Interactive Functions $X \rightarrow Y$

- questions: $(X \rightarrow Y)^- = X^+ + Y^-$
- answers: $(X \rightarrow Y)^+ = X^- + Y^+$
Types of Interaction: $X \rightsquigarrow Y$

**Problem**: There is no addition function

```
add: [int] \rightarrow [int] \rightarrow [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 $A$. 

Exponentials

Game semantics and geometry of interaction use the special case

$$!X := \text{nat} \cdot X$$
Types of Interaction: \( A \cdot X \)

Example

\[
\begin{array}{c}
[A] \rightarrow A \cdot [B] \rightarrow [A \times B] \\
() \\
() \\
(a) \\
(a, ()) \\
(a, b) \\
(a, b)
\end{array}
\]

\[
\begin{aligned}
f(a : A) &= m(a, () ) \\
g(x : A \times B) &= n(x) \\
h(x : \text{unit}) &= k(x)
\end{aligned}
\]
Types of Interaction: $A \cdot X$

**Example**

\[
\left\{(\text{unit} + A) \cdot ([\text{bool}] \rightarrow [A]) \rightarrow [A \times A]\right\}
\]

\[
\begin{align*}
&\text{(inl()},()) \\
&(\text{inl()},()) \\
&(\text{inl()},\text{false}) \\
&(\text{inl()},a_1) \\
&(\text{inr}(a_1),()) \\
&(\text{inr}(a_1),\text{true}) \\
&(\text{inr}(a_1),a_2) \\
\end{align*}
\]
Types of Interaction: $A \cdot X$

Examples

\[
\text{unit} \cdot X \xrightarrow{} X
\]

\[
A \cdot (B \cdot X) \xrightarrow{} (A \times B) \cdot X
\]

\[
(A + B) \cdot X \xrightarrow{} (A \cdot X) \otimes (B \cdot X)
\]
Types of Interaction: \( !X \)

**Exponential** \( !X \)

\[ !X := \text{nat} \cdot X \]

where

\[
\text{type } \text{nat} = \text{Zero of unit} \\
\quad | \quad \text{Succ of nat}
\]

Game Semantics and Geometry of Interaction use this special case.
Retraction $A \triangleright B$

Write $A \triangleright B$ if there exist two program fragments

such that

behave like
Types of Interaction: $!X$

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

$$B \cdot X \rightarrow A \cdot X$$

![Diagram](image-url)
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]

\[(A + B) \cdot X \rightarrow (A \cdot X) \otimes (B \cdot X)\]

\[!_{p+q}X \rightarrow !_pX \otimes !_qX\]

\[enc(inl(v)) = enc(v)\]
\[enc(inr(w)) = p + enc(w)\]

\[!_{2\cdot\max(p,q)}X \rightarrow !_pX \otimes !_qX\]

\[enc(inl(v)) = \langle 0, enc(v) \rangle\]
\[enc(inr(w)) = \langle 1, enc(w) \rangle\]
Types of Interaction: $A \rightarrow X$

Value Passing $A \rightarrow X$

- questions: $(A \rightarrow X)^{-} = A \times X^{-}$
- answers: $(A \rightarrow X)^{+} = X^{+}$

Compare: $[A] \rightarrow X$
Types of Interaction: $\forall \alpha \triangleleft A. X$ and $\exists \alpha \triangleleft A. X$

Value-Type Polymorphism $\forall \alpha \triangleleft A. X$
- questions: $(\forall \alpha \triangleleft A. X)^- = (\exists \alpha \triangleleft A. X)^- = X^- [A/\alpha]$
- answers: $(\forall \alpha \triangleleft A. X)^+ = (\exists \alpha \triangleleft A. X)^+ = X^+ [A/\alpha]$

Special cases:

$\forall \alpha. X := \forall \alpha \triangleleft \text{nat. } X$

$\exists \alpha. X := \forall \alpha \triangleleft \text{nat. } X$
Summary

\[ [A]^− = \text{unit} \]
\[ [A]^+ = A \]

\[ (X \to Y)^− = X^+ + Y^− \]
\[ (X \to Y)^+ = X^- + Y^+ \]

\[ (A \to X)^− = A \times X^− \]
\[ (A \to X)^+ = X^+ \]

\[ (A \cdot X)^− = A \times X^− \]
\[ (A \cdot X)^+ = A \times X^+ \]

\[ (\forall α.X)^− = X^−[\text{nat}/α] \]
\[ (\forall α.X)^+ = X^+[\text{nat}/α] \]