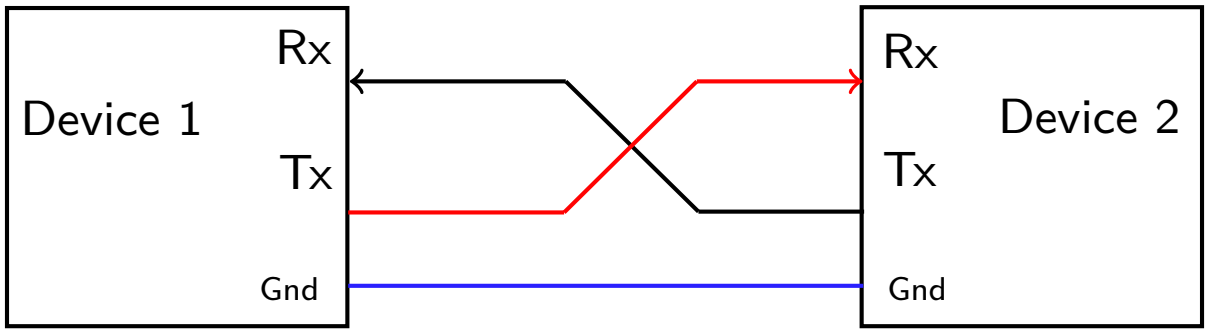
Compositionality and monadic effects

Challenges and perspectives for certification of real-time guarrantes

Benjamin Lion, TEA
Jean-Pierre Talpin, TEA
David Nowak, 2XS CNRS Lille

Running example

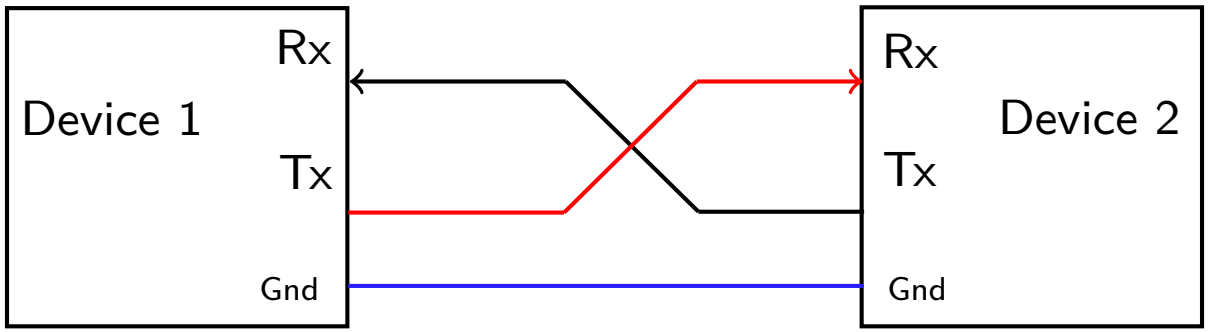
Universal Asynchronous Receiver Transmitter (UART)



- communication between two devices without shared clock (Asynchronous);
- pre-defined speed (baudrates) between Device 1 and Device 2;
- rules to send and receive symbols.

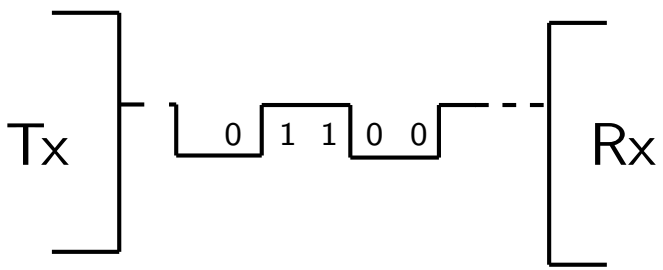
Running example

Universal Asynchronous Receiver Transmitter (UART)



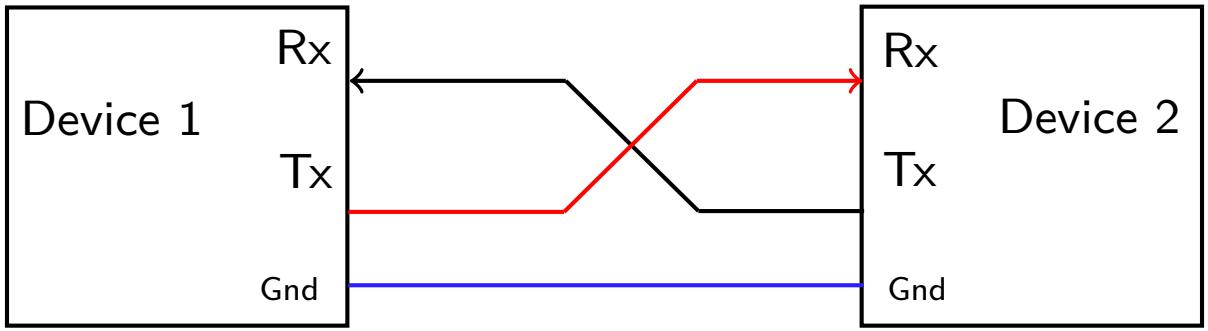
- communication between two devices without shared clock (Asynchronous);
- pre-defined speed (baudrates) between Device 1 and Device 2;
- rules to send and receive symbols.

Example: send the sequence *00110*



Running example

Universal Asynchronous Receiver Transmitter (UART)



- communication between two devices without shared clock (Asynchronous);
- pre-defined speed (baudrates) between Device 1 and Device 2;
- rules to send and receive symbols.

Motivations:

- implement a bit-banging receiver and transmitter (in software) - how fast to sample?;
- verify that the communication follows the UART protocol.

Pseudo-code

Transmitter:

```
baudrate := 9600;
char s[8] :=
  {0,1,1,0,0,0,1,0};
int c := 0;
TxSend(0);
wait 1/baudrate;
while c < 8 do
  TxSend(s[c]);
  wait 1/baudrate;
  c := c+1;
od
```

Pseudo-code

Transmitter:

```
baudrate := 9600;
char s[8] :=
  {0,1,1,0,0,0,1,0};
int c := 0;
TxSend(0);
wait 1/baudrate;
while c < 8 do
  TxSend(s[c]);
  wait 1/baudrate;
  c := c+1;
od
```

Receiver:

```
baudrate := 9600;
char s[];
int c := 0;
while true do
 if RxLow then
  wait 1/baudrate;
  while c < 8 do
    s ++ RxSense();
    wait 1/baudrate;
    c := c+1;
  od od
```

Pseudo-code

Transmitter:

```
baudrate := 9600;
char s[8] :=
  {0,1,1,0,0,0,1,0};
int c := 0;
TxSend(0);
wait 1/baudrate;
while c < 8 do
  TxSend(s[c]);
  wait 1/baudrate;
  c := c+1;
od
```

Receiver:

```
baudrate := 9600;
char s[];
int c := 0;
while true do
 if RxLow then
  wait 1/baudrate;
  while c < 8 do
    s ++ RxSense();
    wait 1/baudrate;
    c := c+1;
  od od
```

- TxSend and RxSense are physical actuation and sensing;

Pseudo-code

Transmitter:

```
baudrate := 9600;
char s[8] :=
  {0,1,1,0,0,0,1,0};
int c := 0;
TxSend(0);
wait 1/baudrate;
while c < 8 do
  TxSend(s[c]);
  wait 1/baudrate;
  c := c+1;
od
```

Receiver:

```
baudrate := 9600;
char s[];
int c := 0;
while true do
 if RxLow then
  wait 1/baudrate;
  while c < 8 do
    s ++ RxSense();
    wait 1/baudrate;
    c := c+1;
  od od
```

- TxSend and RxSense are physical actuation and sensing;
- wait and RxLow are time sensitive instructions.

Pseudo-code

Transmitter:

```
baudrate := 9600;
char s[8] :=
  {0,1,1,0,0,0,1,0};
int c := 0;
TxSend(0);
wait 1/baudrate;
while c < 8 do
  TxSend(s[c]);
  wait 1/baudrate;
  c := c+1;
od
```

Receiver:

```
baudrate := 9600;
char s[];
int c := 0;
while true do
 if RxLow then
  wait 1/baudrate;
  while c < 8 do
    s ++ RxSense();
    wait 1/baudrate;
    c := c+1;
  od od
```

- TxSend and RxSense are physical actuation and sensing;
- wait and RxLow are time sensitive instructions.

How to give a denotational and compositional semantics to prove real time properties of the composition?

Composition

Capture functionally and compositionally the propagation of time, i.e., given two time sensitive C programs P_1 and P_2,

$$[\![P_1 \,|\!>\, P_2]\!] = f([\![P_1]\!], [\![P_2]\!])$$

where

- P_1 and P_2 are executed asynchronously (clock is not necessarily shared);
- the output of P_1 is given as input to P_2.

Composition

Capture functionally and compositionally the propagation of time, i.e., given two time sensitive C programs P_1 and P_2,

$$[\![P_1 \,|\!>\!(N \,|\!>\! P_2)]\!] = f([\![P_1]\!], [\![N]\!], [\![P_2]\!])$$

where

- P_1 and P_2 are executed asynchronously (clock is not necessarily shared);
- N models the physics of the hardware;
- the output of P_1 is given as input to P_2 through N.

Composition

Capture functionally and compositionally the propagation of time, i.e., given two time sensitive C programs P_1 and P_2,

$$[\![P_1 \,|\!>(N \,|\!> P_2)]\!] = f([\![P_1]\!], [\![N]\!], [\![P_2]\!])$$

where

- P_1 and P_2 are executed asynchronously (clock is not necessarily shared);
- N models the physics of the hardware;
- the output of P_1 is given as input to P_2 through N.

Goal:

- define sequential composition and parallel composition via the $|\!>$ operation.

Research plan

1. Compositional semantics
   Real programs have effects, such as time.

Research plan

1. Compositional semantics
   Real programs have effects, such as time.

2. Certified compilation
   Extend semantic preservation theorem (CompCert).

Research plan

1. Compositional semantics
   Real programs have effects, such as time.

2. Certified compilation
   Extend semantic preservation theorem (CompCert).

3. Application
   Reference implementation for UART protocol.

Compositionality

Real programs execute on a physical hardware, and each instruction has an effect (time, energy, actuation, memory, ...).

Compositionality

Real programs execute on a physical hardware, and each instruction has an effect (time, energy, actuation, memory, ...).

The functionality of the program depends on a physical state (e.g., memory read/write).

Compositionality

Real programs execute on a physical hardware, and each instruction has an effect (time, energy, actuation, memory, ...).

The functionality of the program depends on a physical state (e.g., memory read/write).

To get back functional features (composition), extend the nature of a function to include *effects* (e.g., define $f^\dagger : A \to MB$ for $f : A \to B$ where M adds effects to B). Then,

$$\frac{f^\dagger : A \to MB, \qquad g^\dagger : B \to MC}{g^\dagger \circ f^\dagger : A \to MC}$$

Compositionality

Real programs execute on a physical hardware, and each instruction has an effect (time, energy, actuation, memory, ...).

The functionality of the program depends on a physical state (e.g., memory read/write).

To get back functional features (composition), extend the nature of a function to include *effects* (e.g., define $f^\dagger : A \to MB$ for $f : A \to B$ where M adds effects to B). Then,

$$\frac{f^\dagger : A \to MB, \qquad g^\dagger : B \to MC}{g^\dagger \circ f^\dagger : A \to MC}$$

What structure for reactive programs with physical effects, such as time passing?

Compositional semantics

Define reactive processes with denotational semantics of type $WA \to MB$,

[1] The essence of Dataflow Programming, T. Uustalu and V. Vene, 2005
[2] Notions of computations and monads, E. Moggi, 1991
[3] Combining a monad and a comonad, J. Power and H. Watanabe, 2002

Compositional semantics

Define reactive processes with denotational semantics of type $WA \to MB$,
where

- W is a co-monad that models the context of execution[1], i.e., history, real time labels;

[1] The essence of Dataflow Programming, T. Uustalu and V. Vene, 2005
[2] Notions of computations and monads, E. Moggi, 1991
[3] Combining a monad and a comonad, J. Power and H. Watanabe, 2002

Compositional semantics

Define reactive processes with denotational semantics of type $WA \to MB$,
where

- W is a co-monad that models the context of execution[1], i.e., history, real time labels;
- M is a monad that models the effect of a computation[2], i.e., errors, non-determinism, memory changes;

[1] The essence of Dataflow Programming, T. Uustalu and V. Vene, 2005
[2] Notions of computations and monads, E. Moggi, 1991
[3] Combining a monad and a comonad, J. Power and H. Watanabe, 2002

Compositional semantics

Define reactive processes with denotational semantics of type $WA \to MB$,
where

- W is a co-monad that models the context of execution[1], i.e., history, real time labels;
- M is a monad that models the effect of a computation[2], i.e., errors, non-determinism, memory changes;

Composition is given by $[\![P_1 \,|{>}\, P_2]\!] : WA \to MC$, with

- two processes $P_1 : WA \to MB$ and $P_2 : WB \to MC$; and
- the proof of distributivity[3] of W over M.

---

[1] The essence of Dataflow Programming, T. Uustalu and V. Vene, 2005
[2] Notions of computations and monads, E. Moggi, 1991
[3] Combining a monad and a comonad, J. Power and H. Watanabe, 2002

Compositional semantics

Example of compositional semantics for reactive programs:

- with errors: $A^+ \to B + 1$,
  distributive law between $\_^+$ and $\_ + 1$ proved in Coq;
- with non-determinism: $A^+ \to \mathcal{P}(B) \setminus \{\emptyset\}$,
  distributive law between $\_^+$ and $\mathcal{P}(\_) \setminus \{\emptyset\}$ (on-going work).

Compositional semantics

Example of compositional semantics for reactive programs:

- with errors: $A^+ \to B + 1$,
  distributive law between $\_^+$ and $\_ + 1$ proved in Coq;
- with non-determinism: $A^+ \to \mathcal{P}(B) \setminus \{\emptyset\}$,
  distributive law between $\_^+$ and $\mathcal{P}(\_) \setminus \{\emptyset\}$ (on-going work).

As a result, the distributivity law induces a bi-Kleisli category with the following composition:

$$\frac{[\![P_1]\!] : A^+ \to B + 1, \quad [\![P_2]\!] : B^+ \to C + 1}{[\![P_1 \mid\!> P_2]\!] = [\![P_2]\!] \circ [\![P_1]\!] : A^+ \to C + 1}$$

Compositional semantics

Example of compositional semantics for reactive programs:

- with errors: $A^+ \to B + 1$,
  distributive law between $\_^+$ and $\_ + 1$ proved in Coq;
- with non-determinism: $A^+ \to \mathcal{P}(B) \setminus \{\emptyset\}$,
  distributive law between $\_^+$ and $\mathcal{P}(\_) \setminus \{\emptyset\}$ (on-going work).

As a result, the distributivity law induces a bi-Kleisli category with the following composition:

$$\frac{[\![P_1]\!] : A^+ \to B + 1, \quad [\![P_2]\!] : B^+ \to C + 1}{[\![P_1 \mid\!> P_2]\!] = [\![P_2]\!] \circ [\![P_1]\!] : A^+ \to C + 1}$$

Note that:

- induced composition $\circ$ is associative and has an identity;
- we can prove equalities such as $[\![P_1 \mid\!> P_2]\!] = [\![P_2 \mid\!> P_1]\!]$.

Compositional semantics

Let

- $\mathbb{N}^+$ co-monad;
- $\mathbb{N} + 1$ monad;
- processes $P_1, P_2 : \mathbb{N}^+ \to \mathbb{N} + 1$.

Example,

$[\![P_1]\!]$		$[\![P_2]\!]$		$[\![P_2]\!] o [\![P_1]\!]$	
$[0]$	1	$[1]$	0	$[0]$	0

[4] Implementing Hybrid Semantics: From Functional to Imperative, S. Goncharov, R. Neves, J. Proença, 2020

Compositional semantics

Let

- $\mathbb{N}^+$ co-monad;
- $\mathbb{N} + 1$ monad;
- processes $P_1, P_2 : \mathbb{N}^+ \to \mathbb{N} + 1$.

Example,

$[\![P_1]\!]$		$[\![P_2]\!]$		$[\![P_2]\!] o [\![P_1]\!]$	
$[0]$	1	$[1]$	0	$[0]$	0
$[0,0]$	$\bot$			$[0,0]$	$\bot$

[4] Implementing Hybrid Semantics: From Functional to Imperative, S. Goncharov, R. Neves, J. Proença, 2020

Compositional semantics

Let

- $\mathbb{N}^+$ co-monad;
- $\mathbb{N} + 1$ monad;
- processes $P_1, P_2 : \mathbb{N}^+ \to \mathbb{N} + 1$.

Example,

$[\![P_1]\!]$		$[\![P_2]\!]$		$[\![P_2]\!] o [\![P_1]\!]$	
$[0]$	1	$[1]$	0	$[0]$	0
$[0,0]$	$\bot$			$[0,0]$	$\bot$
$[0,0,1]$	1	$[1,1]$	$\bot$	$[0,0,1]$	$\bot$

[4] Implementing Hybrid Semantics: From Functional to Imperative, S. Goncharov, R. Neves, J. Proença, 2020

Compositional semantics

Let

- $\mathbb{N}^+$ co-monad;
- $\mathbb{N} + 1$ monad;
- processes $P_1, P_2 : \mathbb{N}^+ \to \mathbb{N} + 1$.

Example,

$[\![P_1]\!]$		$[\![P_2]\!]$		$[\![P_2]\!] o [\![P_1]\!]$	
$[0]$	1	$[1]$	0	$[0]$	0
$[0,0]$	$\bot$			$[0,0]$	$\bot$
$[0,0,1]$	1	$[1,1]$	$\bot$	$[0,0,1]$	$\bot$
$[0,0,1,1]$	0	$[1,1,0]$	1	$[0,0,1,1]$	1

[4] Implementing Hybrid Semantics: From Functional to Imperative, S. Goncharov, R. Neves, J. Proença, 2020

Compositional semantics

Let

- $\mathbb{N}^+$ co-monad;
- $\mathbb{N} + 1$ monad;
- processes $P_1, P_2 : \mathbb{N}^+ \to \mathbb{N} + 1$.

Example,

$[\![P_1]\!]$		$[\![P_2]\!]$		$[\![P_2]\!] o [\![P_1]\!]$	
$[0]$	1	$[1]$	0	$[0]$	0
$[0,0]$	$\bot$			$[0,0]$	$\bot$
$[0,0,1]$	1	$[1,1]$	$\bot$	$[0,0,1]$	$\bot$
$[0,0,1,1]$	0	$[1,1,0]$	1	$[0,0,1,1]$	1

On-going work:

- choose monad and co-monad to model time propagation[4]; and
- add physical processes to model network delays.

[4] Implementing Hybrid Semantics: From Functional to Imperative, S. Goncharov, R. Neves, J. Proença, 2020

Compilation

Goal: preserve the timing properties through compilation.

State of the art:

- CompCert observable behavior excludes execution time: *everything the user of the program, or the physical world in which it executes, can "see" about the actions of the program, with the notable exception of execution time and memory consumption*;
- WCET analysis is not compositional (cf AbsInt): *the analysis of the worst-case execution time is not compositional: given a composition A;B, the execution time of instruction B may depend on the state of the architecture after execution of instruction A.*

Compilation

Theorem (Semantic preservation)

$bh' \in ASem(tp) \Rightarrow \exists bh.bh \in CSem(p) \wedge bh \subseteq bh'$

where:

- *ASem* is semantics of *A*ssembly (target language), and *CSem* is semantics of *C* language (input language);
- $\subseteq$ means bh' improves on bh (i.e., undefined behavior in bh are defined in bh').

Compilation

Theorem (Semantic preservation)

$bh' \in ASem(tp) \Rightarrow \exists bh.bh \in CSem(p) \wedge bh \subseteq bh'$

where:

- *ASem* is semantics of *A*ssembly (target language), and *CSem* is semantics of *C* language (input language);
- $\subseteq$ means bh' improves on bh (i.e., undefined behavior in bh are defined in bh').

Theorem (Semantic preservation')

$bh' \in ASem'(tp) \Rightarrow \exists bh.bh \in CSem'(p) \wedge bh \subseteq_{\prime} bh'$

where

- *ASem* and *CSem* extend $ASem'$ and $CSem'$ to include timing information, using (co-)monadic semantics;
- $\subseteq_{\prime}$ refines the behavior with physical models and timing information.

Application

White Rabbit experimentally synchronizes clocks at precision below **nano-seconds**. Can we provably certify that such precision holds?

Three layers to formalize:

- physical layer: time labels are generated by Dual Mixer Time Difference (DMTD).
- link layer: the Layer-1 Synchronization (SyncE) algorithm extracts a clock frequency from a flow of time stamped messages.
- network layer: Precise Time Protocol (PTP) synchronises a network with a distributed algorithm. PTP requires a precise source of time, and synchronizes nodes on the network with a precision below nanosecond, in the case of White Rabbit.

Future work: Type system

Logic to specify rules of time sensitive protocols (e.g., UART).

Related work: LTL types FRP[5], Hybrid Pi-calculus with session types (ISCAS collaboration).

[5] LTL types FRP, A. Jeffrey, 2012

Future work: Type system

Logic to specify rules of time sensitive protocols (e.g., UART).

Related work: LTL types FRP[5], Hybrid Pi-calculus with session types (ISCAS collaboration).

Enforcing properties statically, i.e., prove that

$$\vdash P_1 \,|{>}\, N \,|{>}\, P_2 : \phi_{UART}$$

where

- P_1 and P_2 are running programs on the Device 1 and 2 respectively;
- N specifies the physics of the hardware;
- ϕ_{UART} logically captures the specification of the UART protocol;
- PAT correspondence: Protocols As Types, Proofs As Terms.

[5] LTL types FRP, A. Jeffrey, 2012

Conclusion

The logic of a program can be preempted by physical effects.

Conclusion

The logic of a program can be preempted by physical effects.

Need for new theoretical framework to unify physics within notion of computations.

Conclusion

The logic of a program can be preempted by physical effects.

Need for new theoretical framework to unify physics within notion of computations.

Existing work on effects and functional programming seems to be a good starting point.