Implementation and Verification of Modular Effectful Systems in Coq using FreeSpec

Yann Régis-Gianas
(IRIF, Univ. Paris Diderot, Inria) – yrg@irif.fr

with **Thomas Letan** (ANSSI), Vincent Tourneur (Inria, IRIF),
Pierre Chifflier (ANSSI) and Guillaume Hiet (CentraleSupélec, Inria)

2019-09-03 – PPS days

A classic IRC joke about Coq programmers

```
someone> Could you write a certified compiler for me?
coq-der> My pleasure!
```

A classic IRC joke about Coq programmers

```
someone> Could you write a certified compiler for me?
coq-der> My pleasure!

someone> Could you write a hello world program for me?
coq-der> No.
-!-      coq-der [~xl@166.37.73.42] has left #coq [Crying]
```

A classic IRC joke about Coq programmers

```
someone> Could you write a certified compiler for me?
coq-der> My pleasure!

someone> Could you write a hello world program for me?
coq-der> No.
-!-      coq-der [~xl@166.37.73.42] has left #coq [Crying]
```

This joke does not work anymore.

Let's make Gallina
a general purpose programming language!

This talk

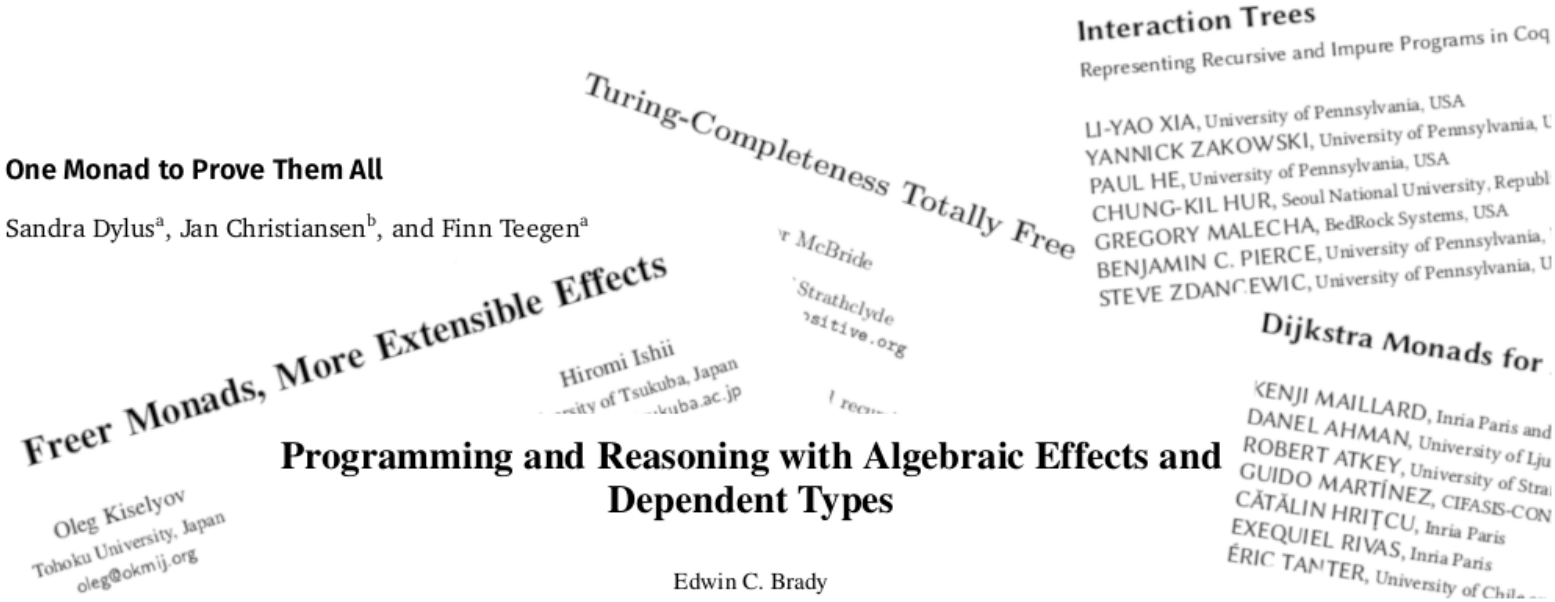
FreeSpec is a Coq library and plugin to develop effectful systems in Gallina

What are the design choices of FreeSpec?

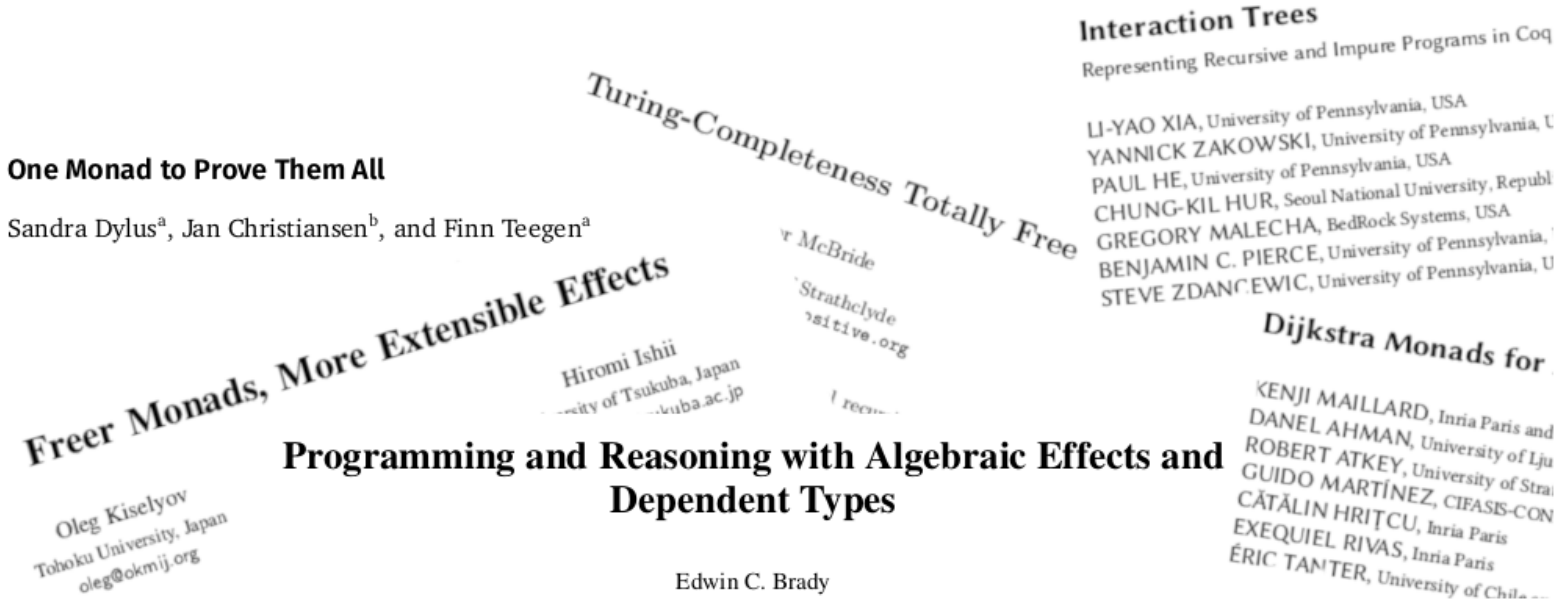
- How to write **effectful programs**?
- How to **build large systems** by composition of effectful components?
- How to **specify them**?
- How to **reason about them**?
- What are the **limitations** of FreeSpec?

How to write effectful programs in Gallina?

A rich design space of representations for effecful programs



A rich design space of representations for effecful programs



FreeSpec focuses on
Modularity of implementations, specifications and proofs.

Effectful operations as interfaces

Interface: a collection of effects with heterogeneous types of answers.

Effectful operations as interfaces

Interface: a collection of effects with heterogeneous types of answers.

```
1  Definition Interface := Type -> Type.
```

Effectful operations as interfaces

Interface: a collection of effects with heterogeneous types of answers.

```
1  Definition Interface := Type -> Type.
```

Example: **Interface for basic terminal interaction.**

```
1  Inductive i: Type -> Type :=
2  | Scan: i string
3  | Echo: string -> i unit.
```

Effectful operations as interfaces

Interface: a collection of effects with heterogeneous types of answers.

```
1  Definition Interface := Type -> Type.
```

Example: **Interface for file manipulation.**

```
1  Inductive i: Type -> Type :=
2  | Stat: string -> i stats
3  | Open: mode -> options -> string -> i fd
4  | OpenDir: string -> i fd
5  | FStat: fd -> i stats
6  | GetSize: fd -> i N
7  | Read: N -> fd -> i string
8  | ReadDir: fd -> i string
9  | Write: string -> fd -> i unit
10 | Seek: seekRef -> fd -> fd -> i unit
11 | Close: fd -> i unit
12 | CloseDir: fd -> i unit.
```

Programs as inhabitants of a free monad

A **program** either purely computes or it interacts with its environment.

```
1  Inductive Program (I:  Interface) (A:  Type) :=
2  | Pure (a:  A) : Program I A
3  | Request {B:  Type} (e:  I B) (f:  B -> Program I A) : Program I A.
```

Programs as inhabitants of a free monad

A **program** either purely computes or it interacts with its environment.

```
Inductive Program (I: Interface) (A: Type) :=
| Pure (a: A) : Program I A
| Request {B: Type} (e: I B) (f: B -> Program I A) : Program I A.
```

Example

```
Definition hello : Program Console.i unit :=
  Request Console.Scan (fun name =>
  Request (Console.Echo ("Hello " ++ name)) (fun _ =>
  Pure tt)).
```

Programs as inhabitants of a free monad

A **program** either purely computes or it interacts with its environment.

```
1  Inductive Program (I: Interface) (A: Type) :=
2  | Pure (a: A) : Program I A
3  | Request {B: Type} (e: I B) (f: B -> Program I A) : Program I A.
```

Example

```
1  Definition hello {ix} `{Use Console.i ix} : Program ix unit :=
2    name <- scan;
3    echo ("Hello " ++ name).
```

A larger example

Let us have a look at the implementation of **`coqar`**,
a version of **`ar`** implemented using FreeSpec.[1]

[1] Work-in-progress with Vincent Tourneur and Thomas Letan.

An archive produced by `ar`

An `ar` file is a textual representation of the concatenation of several files:

```
1  !<arch>
2  test1.txt/      0           0     0     644     14        `
3  coqar example
4  test2.txt/      0           0     0     644     8         `
5  bonjour
```

ar in Coq

```
1  Definition create {ix} `{Use FileSystem.i ix}
2    (files : list string) (output : string)
3  : Program ix unit :=
4    fd <- open WriteOnly MayCreateTruncate output;
5    write_header fd;;
6    insert_files fd files;;
7    close fd.
```

ar in Coq

```
1  Fixpoint insert_files {ix} `{Use FileSystem.i ix}
2    (fd : fd) (files : list string)
3  : Program ix unit :=
4    match files with
5    | file :: l =>
6      write_entry fd file;;
7      insert_files fd l
8    | _ => pure tt
9    end.
```

ar in Coq

```
1  Definition write_entry {ix} `{Use FileSystem.i ix}
2    (fd : fd) (name : string)
3  : Program ix unit :=
4    fd2 <- open ReadOnly DontCreate name;
5    size <- getSize fd2;
6    content <- read size fd2;
7    close fd2;;
8    write (gen_header name 644 size) fd;;
9    write new_line fd;;
10   write content fd.
```

ar in Coq

```
1  Definition write_entry {ix} `{Use FileSystem.i ix}
2    (fd : fd) (name : string)
3  : Program ix unit :=
4    fd2 <- open ReadOnly DontCreate name;
5    size <- getSize fd2;
6    content <- read size fd2;
7    close fd2;;
8    write (gen_header name 644 size) fd;;
9    write new_line fd;;
10   write content fd.
```

Could ~~Haskell~~Coq become
the world's finest imperative language?

Coinductive interpretation of programs

- As is, a term of type `Program I A` is like an empty shell.
- We must give a meaning to each effectful operation.
- This interpretation is potentially modified after each effect.
- Hence, this updated interpretation must be returned.

```
1  CoInductive Semantics : Type :=
2  | handler (f: forall {A: Type}, I A -> Result A): Semantics
3
4  with Result: Type -> Type :=
5  | mkRes {A}: A -> Semantics -> Result A.
```

- This idea comes from the Haskell `operational` package.
- The "stream-like" type of `handler` imposes a coinductive definition.

Stateful effect handler

- Semantics are easily equipped with an evolving state.

```
1  Definition PS {I : Interface} (State : Type) :=
2    forall (A: Type), State -> I A -> (A * State).
3
4  CoFixpoint mkSemantics {I : Interface} {State : Type}
5    (ps : PS State) (s : State) : Semantics I :=
6    handler (fun (A:  Type) (e:  I A) =>
7      mkRes (fst (ps A s e)) (mkSemantics ps (snd (ps A s e)))).
```

A strength of FreeSpec : `Program` evaluation

- `Program` evaluation is defined by induction.

```
1  Fixpoint runProgram
2           {I:    Interface}
3           {A:    Type}
4           (sig:  Semantics I)
5           (p:    Program I A)
6    : Result I A :=
7    match p with
8    | Pure a =>
9      mkResult a sig
10   | Request e f =>
11     let res := handle sig e in
12     runProgram (Semantics.next res) (f (Semantics.result res))
13   end.
```

How to realize such an **effect handler**?

How to realize such an **effect handler**?

By simulation – By extraction – By interpretation – By delegation

Realization of semantics : by simulation

- A Coq function can act as a denotation for any semantics.

```
Inductive NatStack : Type -> Type :=
| Push (x: nat) : NatStack unit
| Pop : NatStack (option nat).
```

```
Definition nat_stack_semantics : Semantics NatStack :=
  mkSemantics (fun (A:  Type) (l:  list nat) (e:  NatStack A) =>
     match e with
     | Push x => (tt, x :: l)
     | Pop => match l with
              | x :: r => (Some x, r)
              | _ => (None, nil)
              end
      end) nil.
```

Realization of semantics : by extraction

- Program evaluators extract to straightforward OCaml code.
- Impure functions serve as effective semantics for effecful operations.

```
Fixpoint pipe {ix} `{Use Console.i ix} n : Program ix unit :=
  match n with
  | O => pure tt
  | S k => Console.scan >>= Console.echo;; pipe k
  end.

Axiom ocaml_scan : unit -> string.
Axiom ocaml_echo : string -> unit.

CoFixpoint ocaml_semantics :=
  handler (fun {A} (x : Console.i A) =>
    match x with
    | Console.Scan => Sem.mkRes (ocaml_scan tt) ocaml_semantics
    | Console.Echo s => Sem.mkRes (ocaml_echo s) ocaml_semantics
    end).

Definition run_pipe : unit := evalProgram ocaml_semantics (pipe 10).

Extraction "pipe.ml" run_pipe.
```

Realization of semantics : by interpretation

Or else...

```
1  Fixpoint pipe {ix} `{Use Console.i ix} n: Program ix unit :=
2    match n with
3    | O => pure tt
4    | S k => Console.scan >>= Console.echo;; pipe k
5    end.
6
7  Exec (pipe 10).
```

Realization of semantics : by interpretation

What is `Exec`?

- **`Exec`** is a plugin written in OCaml inspired by the Mtac interpreter.
- It interprets any `Program I A` directly (no extraction or compilation).

Realization of semantics : by interpretation

What is `Exec`?

- **`Exec`** is a plugin written in OCaml inspired by the Mtac interpreter.
- It interprets any **`Program I A`** directly (no extraction or compilation).

Bi-interpreter

Let **`t`** be of type **`Program I A`**.

1. Compute **`w`**, the weak-head normal form of **`t`**.
2. If **`w`** is **`Pure u`**, returns **`u`**.
3. If **`w`** is **`Request e f`**, pass the normal form **`c`** of **`e`** to an effect handler **`g`** written in OCaml, go back to (1) with **`t = f (g c)`**.

How to compose large systems made of effectful components?

Realization of semantics : by delegation

Modularity of implementations:

Your handler is someone else's component
(and conversely).

Realization of semantics : by delegation

Modularity of implementations:

Your handler is someone else's component
(and conversely).

```
1  (* I is the interface implemented by the component. *)
2  (* J is the dependency of the component. *)
3  Definition Component (I:  Interface) (S:  Type) (J:  Interface) :=
4    forall (A: Type), I A -> S -> Program J A * S.
```

Realization of semantics : by delegation

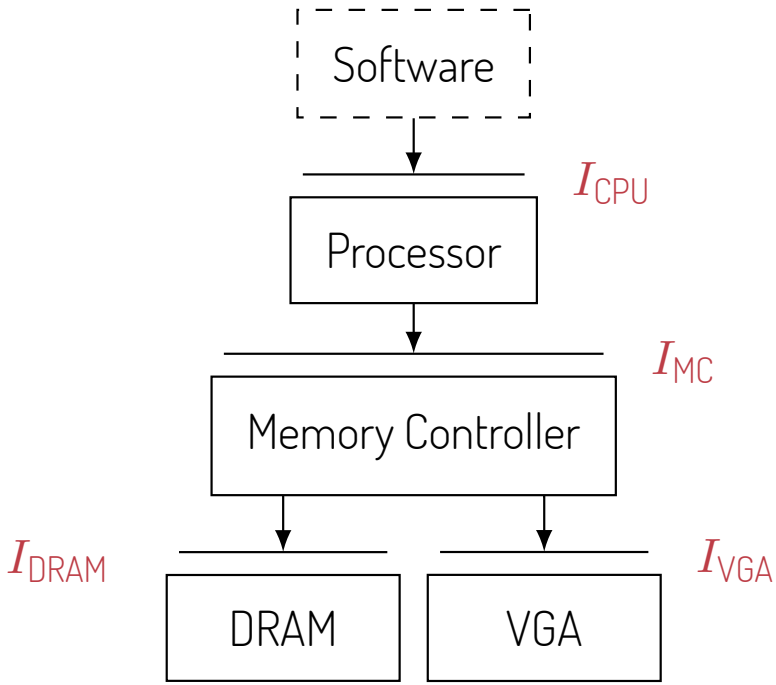
Modularity of implementations:

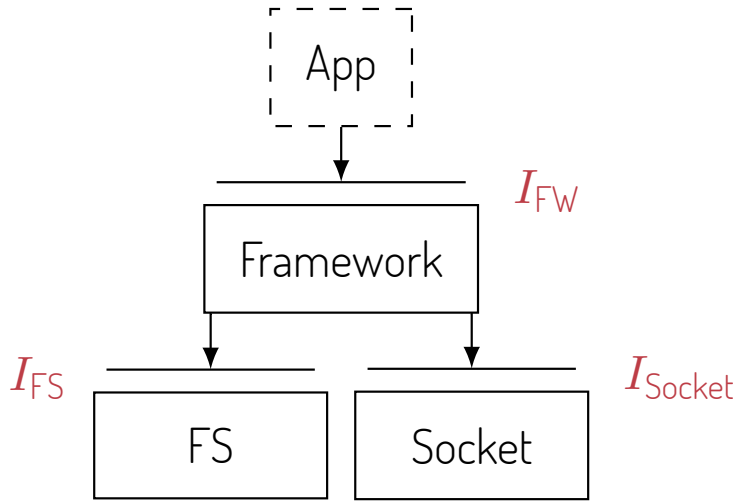
Your handler is someone else's component
(and conversely).

```
(* I is the interface implemented by the component. *)
(* J is the dependency of the component. *)
Definition Component (I:  Interface) (S:  Type) (J:  Interface) :=
  forall (A: Type), I A -> S -> Program J A * S.
```

```
Definition ComponentSemantics {I J:  Interface} {S: Type}
           (c: Component I S J) (s: S) (sig: Semantics J)
: Semantics I :=
  mkSemantics (fun {A: Type} (s':  (S * Semantics J)) (e: I A) =>
    (* ... Implement interface I in terms of interface J ... *)
```

Visual examples





Composing interfaces

Modularity of interfaces:

```
1  Inductive ComposedInterface (I J: Interface) (A: Type) : Type :=
2  | InL (e: I A) : ComposedInterface I J A
3  | InR (e: J A) : ComposedInterface I J A.
4
5  Infix "<+>" := (ComposedInterface) (at level 50, left associativity).
```

```
1  CoFixpoint mkCompSemantics {I J: Interface}
2  (sig_i:  Semantics I) (sig_j:  Semantics J) : Semantics (I <+> J)
3  := (* ... *)
```

How to specify large systems made of effectful components?

Interface specification

- Since components are stateful, specifications must refer to a state.
- To preserve encapsulation, we use a notion of **abstract state**.
- An interface specification over abstract state type `W` is:

```
1  Record InterfaceSpecification (W : Type) (I : Interface) : Type :=
2  {
3    (* An interpreter for effects over abstract states. *)
4    abstract_step : W -> forall A, I A -> A -> W;
5
6    (* The effects that are compatible with a given abstract state. *)
7    allowed_operation : W -> forall A, I A -> Prop;
8
9    (* The effect answers that can arise in a given abstract state.  *)
10   expected_answer : W -> forall A, I A -> A -> Prop
11 }.
```

An abstract state for the FileSystem interface

```
1  Definition partial (A B : Type) := A -> option B.
2
3  Definition fdContent := partial fd ascii.
4
5  Record fdState : Type := MkFdState
6  {
7    mode    : FileSystem.mode;
8    kind    : option FileSystem.fileKind;
9    size    : option nat;
10   pos     : option nat;
11   content : fdContent;
12 }.
13
14 Definition state := partial fd fdState.
```

This abstract state represents partial information about the file system.

An abstract step function for the FileSystem interface

Here is the case for **`Open`**:

```
1  abstract_step (FileSystem.Open m o str) fd s :=
2    let size :=
3      match o with
4      | DontCreateTruncate | MayCreateTruncate | MustCreate => Some 0
5      | _ => None
6      end
7    in
8    setFun s fd {|
9       mode    := m;
10      kind    := None;
11      size    := size;
12      pos     := Some 0
13      content := const None;
14   |};
15 (* ... *)
```

This function refines the static knowledge about the file system.

What is a good abstraction specification for `FileSystem`?

Designing specifications is hard

- This specification of the `FileSystem` interface is simple.
- Simplicity makes the reviewing of proof assumptions tractable.
- Yet, the path is short from "simple" to "simplistic".

FreeSpec's philosophy

- FreeSpec allows multiple specifications for a single interface.
- One can check that these specifications are not contradictory.

Compositionality of abstract specifications

Modularity of specifications

```
1  Theorem compose_specifications {W_I W_J: Type} {I J: Interface}
2    (c_i: Specification W_I I)
3    (c_j: Specification W_J J)
4  : Specification (W_I * W_J) (I <+> J).
5
6  Theorem expand_specification_left {W: Type} {I: Interface}
7    (c:  Specification W I)
8    (J:  Interface)
9  : Specification W (I <+> J).
10
11 Theorem expand_specification_right {W: Type} {J: Interface}
12   (c:  Specification W J)
13   (I:  Interface)
14 : Specification W (I <+> J).
```

How to prove large systems made of effectful components?

Two key notions : Compliance and Respectfulness

Semantics compliance

The coinductive predicate `semantics |= s[w]` means
"For every operation of the `semantics` interface which is allowed by specification `s` in abstract state `w`, the `semantics` only produces `expected_answer`s, and stays compliant on the next abstract step."

Program respectfulness

The inductive predicate `p |> s[w]` means
"The program `p` only performs operations that are allowed by the specification `s` in abstract state `w` and for every valid answers, the continuation of the program stays respectful on the next abstract step."

Component correctness

The two sides of the component verification problem

To be correct, a `Component I S J`:

- must be respectful of `J`, and
- must fulfill the requirements of `I`.

Difficulty

- The abstract states for `I` and `J` are interdependent.
- Their relationship is crucial for the correctness proof of the component.
- Hence the need for a **synchronization predicate**:

```
1  Definition sync_pred (W_I W_J: Type) (S: Type) :=
2    W_I -> S -> W_J -> Prop.
```

Component respectfulness

Informally

A component is **respectful** if it uses its dependencies in a respectful way when it is itself used with respect.

Component respectfulness

Informally

A component is **respectful** if it uses its dependencies in a respectful way when it is itself used with respect.

```
1  Definition respectful_component
2             {W_I W_J:   Type}
3             {I J:       Interface}
4             {S:         Type}
5             (component: Component I S J)
6             (master:    Specification W_I I)
7             (slave:     Specification W_J J)
8             (sync:      sync_pred W_I W_J S) :=
9    forall (w_i: W_I) (s: S) (w_j: W_J) {A: Type} (e: I A),
10     sync w_i s w_j
11     -> allowed_operation master e w_i
12     -> (component A e s) |> slave[w_j].
```

Reasoning principle for Component Compliance

Informally

If the synchronization predicate stays valid during component's execution and is strong enough to imply the expectations about effects' answers, then the component will be compliant when composed with any compliant semantics.

Reasoning principle for Component Compliance

Informally

If the synchronization predicate stays valid during component's execution and is strong enough to imply the expectations about effects' answers, then the component will be compliant when composed with any compliant semantics.

```
Theorem compliant_component {W_I W_J: Type} {I J: Interface} {S: Type}
  (component: Component I S J)
  (master: Specification W_I I)
  (slave: Specification W_J J)
  (sync: sync_pred W_I W_J S)
  (Hsyncpres: sync_preservation component master slave sync)
  (Hsyncp: sync_postcondition component master slave sync)
  (Hrespectful: respectful_component component master slave sync)
  : forall (w_i:   W_I)
           (s:    S)
           (w_j:   W_J)
           (sig:  Sem.t J)
           (Hcomp: sig |= slave[w_j]),
    sync w_i s w_j
    -> ComponentSemantics component s sig |= master[w_i].
```

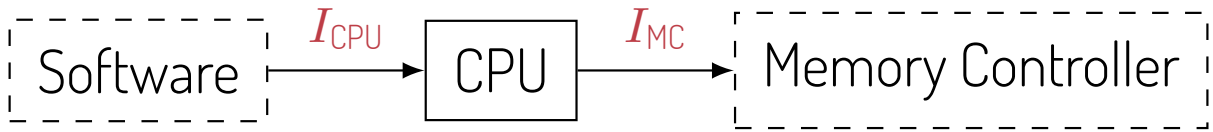
Verification Methodology

Verifying the CPU

Verification Methodology

Verifying the CPU

The CPU executes **software** thanks to a **memory controller**.

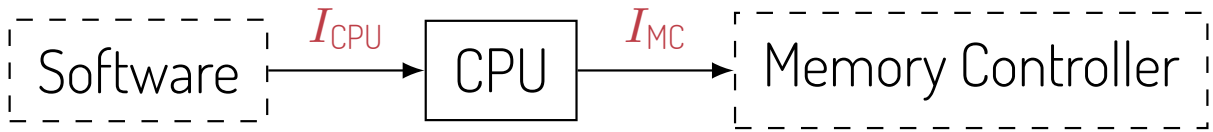


$$\frac{I_{\text{CPU}} \quad CPU \quad I_{\text{MC}}}{\phantom{I_{\text{CPU}} \quad CPU \quad I_{\text{MC}}}}$$

Verification Methodology

Verifying the CPU

We want to prove the CPU complies with an interface specification $(\mathbb{A}, \mathbb{X})$.



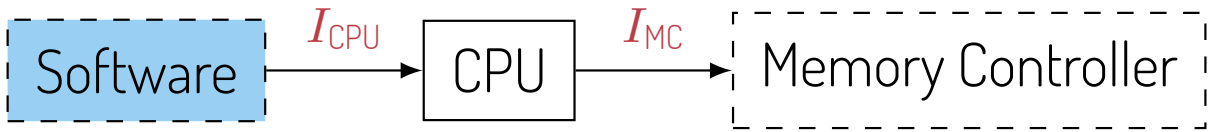
$$\frac{I_{\text{CPU}} \quad \xrightarrow{CPU} \quad I_{\text{MC}}}{\mathbb{A}}$$

$$\mathbb{X}$$

Verification Methodology

Verifying the CPU

We assume the software component will be respectful.



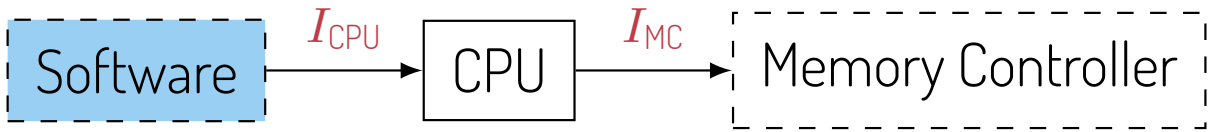
$$\frac{I_{\text{CPU}} \quad \xrightarrow{CPU} \quad I_{\text{MC}}}{\mathbb{A}}$$

$$\mathbb{X}$$

Verification Methodology

Verifying the CPU

We want to verify that, according to the CPU, the expectations $\mathbb{X}$ holds.

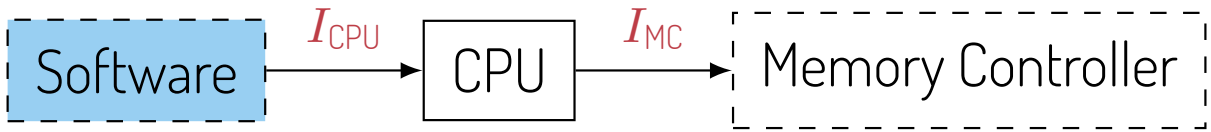


$$\frac{I_{\text{CPU}} \quad \xrightarrow{CPU} \quad I_{\text{MC}}}{\mathbb{A} \Downarrow \mathbb{X}}$$

Verification Methodology

Verifying the CPU

Rather than relying on the memory controller model, we'd rather identify a sufficient couple $(\mathbb{A}', \mathbb{X}')$, and abstract away the memory controller.

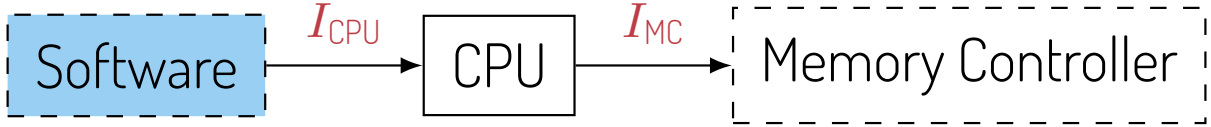


$$\begin{array}{ccc} I_{\text{CPU}} & \xrightarrow{CPU} & I_{\text{MC}} \\ \hline\hline \mathbb{A} & & \mathbb{A}' \\ \\ \mathbb{X} & & \mathbb{X}' \end{array}$$

Verification Methodology

Verifying the CPU

We prove that, because $\mathbb{A}$ is assumed and thanks to the CPU model, then the assumptions $\mathbb{A}'$ are met.



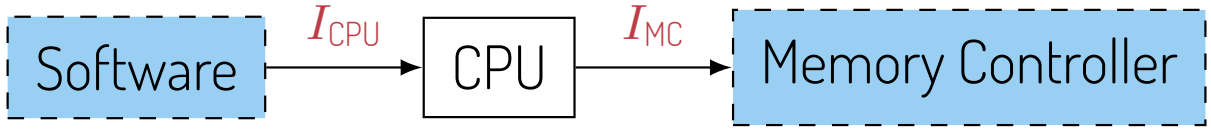
$$\frac{I_{\text{CPU}} \quad \xrightarrow{CPU} \quad I_{\text{MC}}}{\mathbb{A} \quad \Rightarrow \quad \mathbb{A}'}$$

$$\mathbb{X} \qquad\qquad \mathbb{X}'$$

Verification Methodology

Verifying the CPU

We assume the memory controller fulfills the expectations $\mathbb{X}'$.

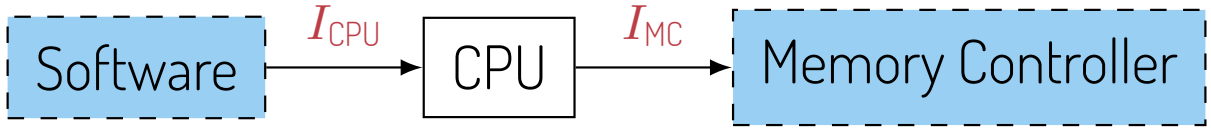


$$\frac{I_{\text{CPU}} \quad \xrightarrow{CPU} \quad I_{\text{MC}}}{\begin{array}{ccc} \mathbb{A} & \Rightarrow & \mathbb{A}' \\ & & \Downarrow \\ \mathbb{X} & & \mathbb{X}' \end{array}}$$

Verification Methodology

Verifying the CPU

We prove that, because $\mathbb{X}'$ and thanks to CPU model, then $\mathbb{X}$ is verified.

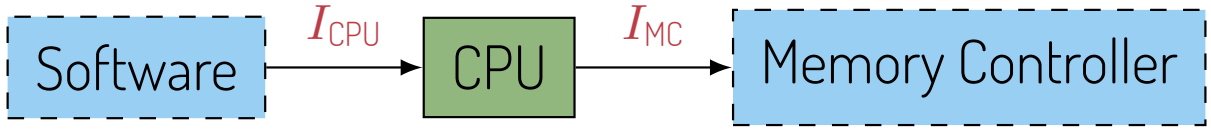


$$\begin{array}{ccc} I_{\text{CPU}} & \xrightarrow{CPU} & I_{\text{MC}} \\ \hline \mathbb{A} & \Rightarrow & \mathbb{A}' \\ & & \Downarrow \\ \mathbb{X} & \Leftarrow & \mathbb{X}' \end{array}$$

Verification Methodology

Verifying the CPU

In other words, our CPU model enforces $\mathbb{X}$ as long as the software component complies to $\mathbb{A}$.

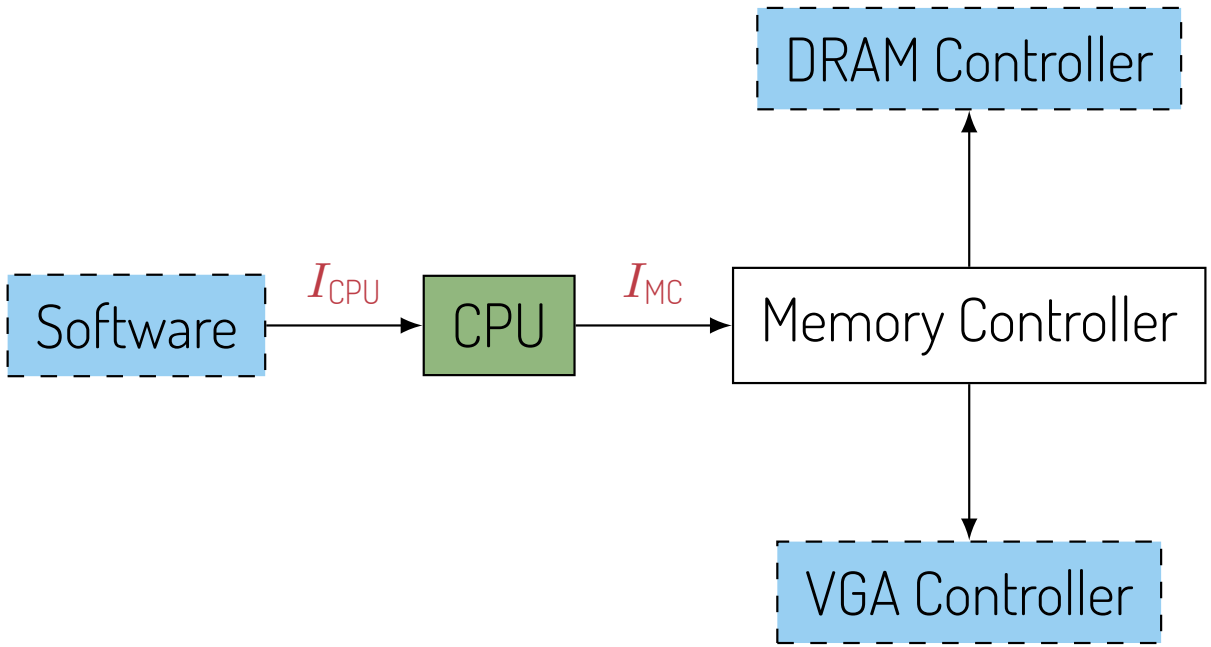


$$\begin{array}{ccc} I_{\text{CPU}} & \xrightarrow{CPU} & I_{\text{MC}} \\ \hline \mathbb{A} & \Rightarrow & \mathbb{A}' \\ \Downarrow & & \Downarrow \\ \mathbb{X} & \Leftarrow & \mathbb{X}' \end{array}$$

Verification Methodology

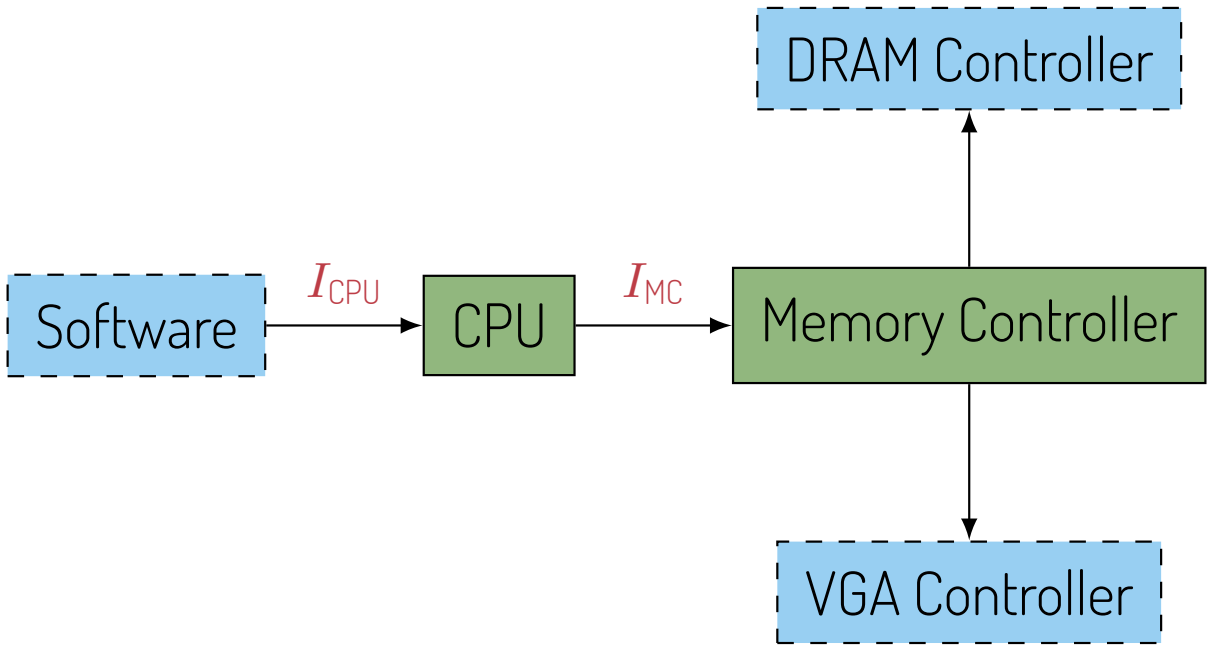
Composing Verification Results

We can follow a similar proof scheme with the memory controller model.



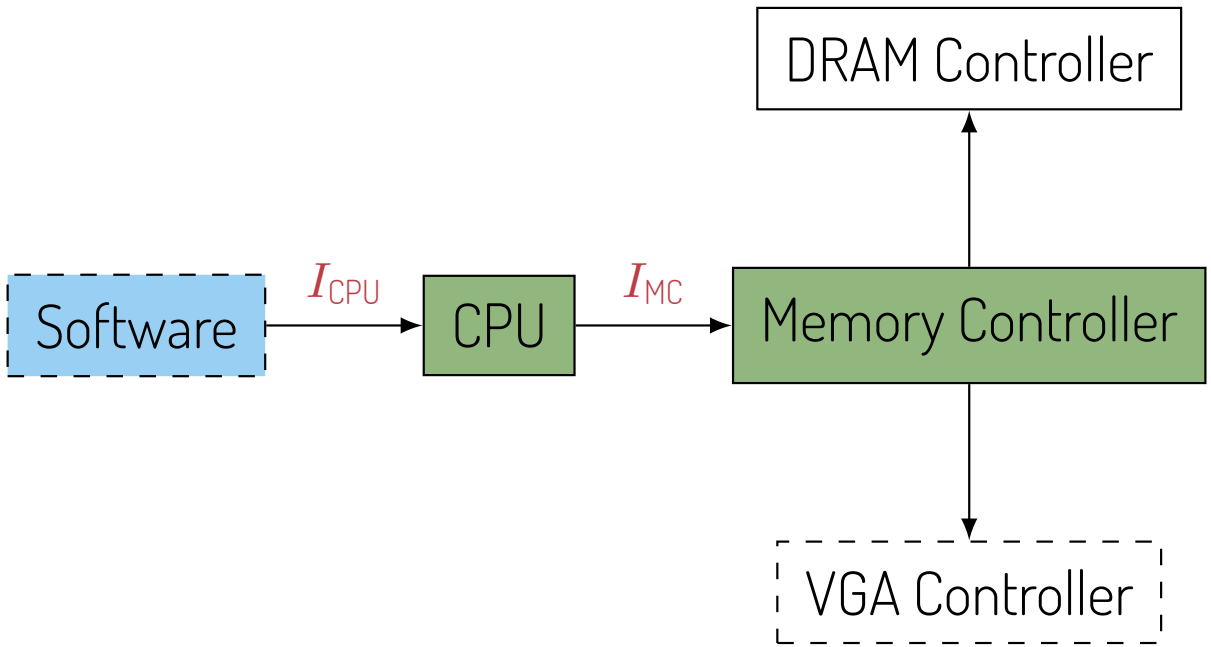
Verification Methodology

Composing Verification Results



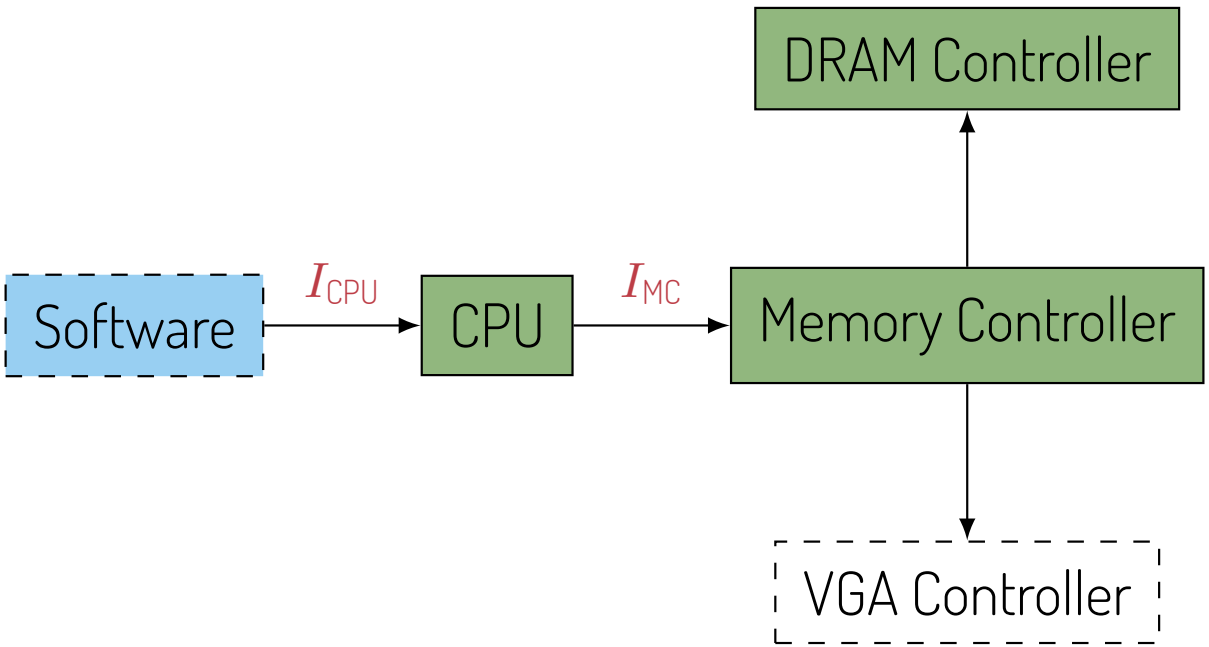
Verification Methodology

Composing Verification Results



Verification Methodology

Composing Verification Results



Theorems about `coqar`

```
1  Definition ar_id {ix} `{Use FileSystem.i ix}
2    (files : list string) (ar : string) : Program ix unit :=
3    create files ar;;
4    remove_files files;;
5    extract ar;
6
7  Theorem ar_spec
8    : ar_id files_names output |> specs[initial].
9
10 Theorem ar_correct (final : state) (r : unit)
11   : correct_run specs initial (ar_id file_names output) r final
12     -> sameFileContents initial final.
```

What are the limitations of FreeSpec?

General recursion

- Interaction trees[2] are coinductively defined:

```
1  CoInductive itree (E : Type -> Type) (R : Type) : Type :=
2  | Ret (r : R)
3  | Tau (t: itree E R)
4  | Vis {A : Type} (e : E A) (k : A -> itree E R).
```

  ... which allows for the representation of diverging computations.
- We could probably patch `Program` to match this definition.
- This would probably increase the complexity of FreeSpec.

[2] DeepSpec

General recursion

- Interaction trees[2] are coinductively defined:

```
1  CoInductive itree (E : Type -> Type) (R : Type) : Type :=
2  | Ret (r : R)
3  | Tau (t: itree E R)
4  | Vis {A : Type} (e : E A) (k : A -> itree E R).
```

  ... which allows for the representation of diverging computations.
- We could probably patch `Program` to match this definition.
- This would probably increase the complexity of FreeSpec.
- We prefer to keep the terminating-by-default behavior ...

  ...seeing **divergence as an effect** per se.

[2] DeepSpec

Looping as an effect

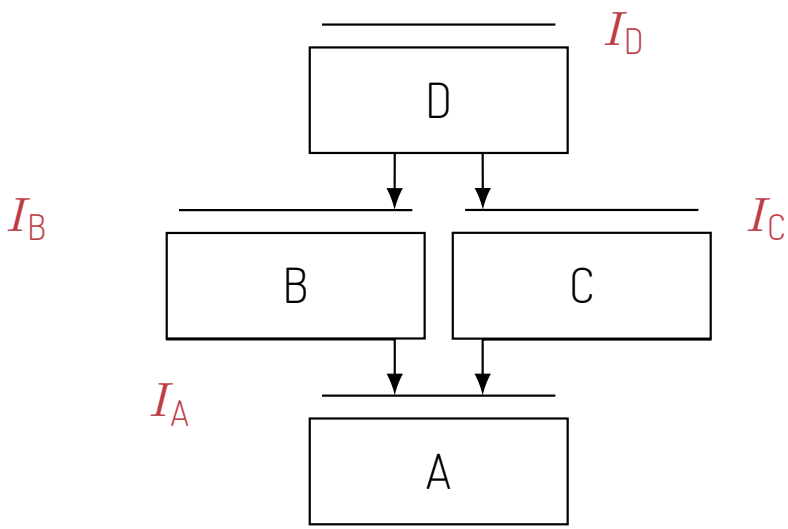
```
1  Inductive interruption A B :=
2  | paused : A -> interruption A B
3  | exited : B -> interruption A B.
4
5  Inductive loop I A B : Interface :=
6  | Loop :
7    (* Initial state *) A
8    (*   Loop body   *) -> (A -> Program I (interruption A B))
9    -> i I A B (option B).
```

- The type `option B` witnesses the fact that a loop may diverge.
- A semantics for `loop` can accumulate the stream of states.
- Contrary to `ITree`, FreeSpec has no support for reasoning about these potentially infinite computations.
- Designing reasoning principles about control operators is future work.

Shared state

Problem

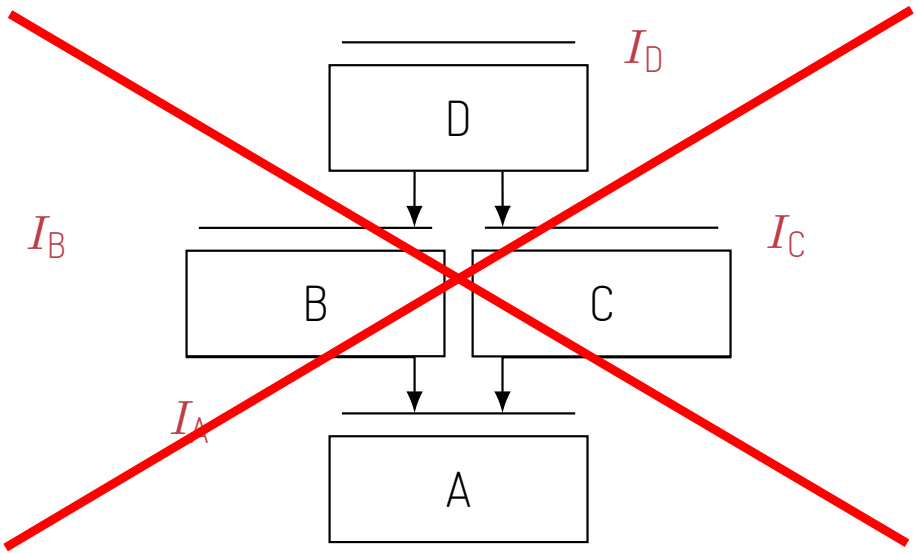
If two components depend on the same component, a diamond appears.

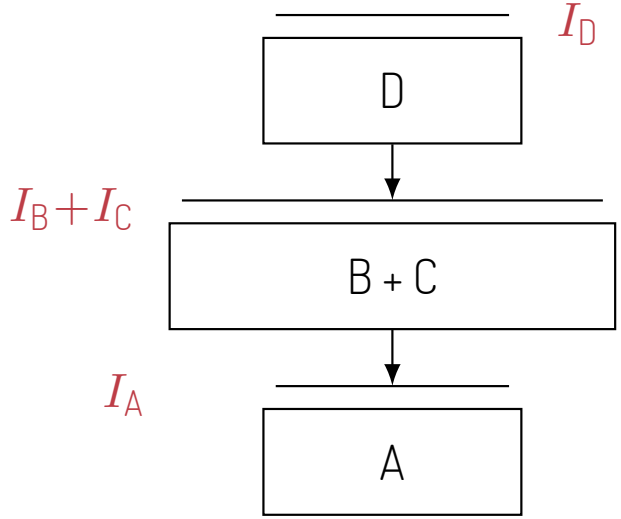


Shared state

Problem

If two components depend on the same component, a diamond appears.





Efficiency

- Exec is a nice convenience but is not tuned for performance.
- Will CertiCoq save us from this inefficiency?
- We need partial evaluation to remove the effect-interpretation layer.

Conclusion

FreeSpec, in a nutshell

FreeSpec

- uses free monads to represent effectful programs in Coq ;
- is able to run effectful programs using **`Exec`** ;
- specifies components with abstract specifications ;
- offers reasoning principles to prove them correct ;
- in a modular way.

Have a look at our FM'18 paper and at the released code :

`http://www.github.com/ANSSI-FR/FreeSpec`

FreeSpec, in a nutshell

FreeSpec

- uses free monads to represent effectful programs in Coq ;
- is able to run effectful programs using `Exec` ;
- specifies components with abstract specifications ;
- offers reasoning principles to prove them correct ;
- in a modular way.

Have a look at our FM'18 paper and at the released code :

`http://www.github.com/ANSSI-FR/FreeSpec`

Thank you for your attention! Questions?