Synthesis of Certified Programs with Effects Using Monads in Coq

Sara Fabbro ▲ Marino Miculan
University of Udine

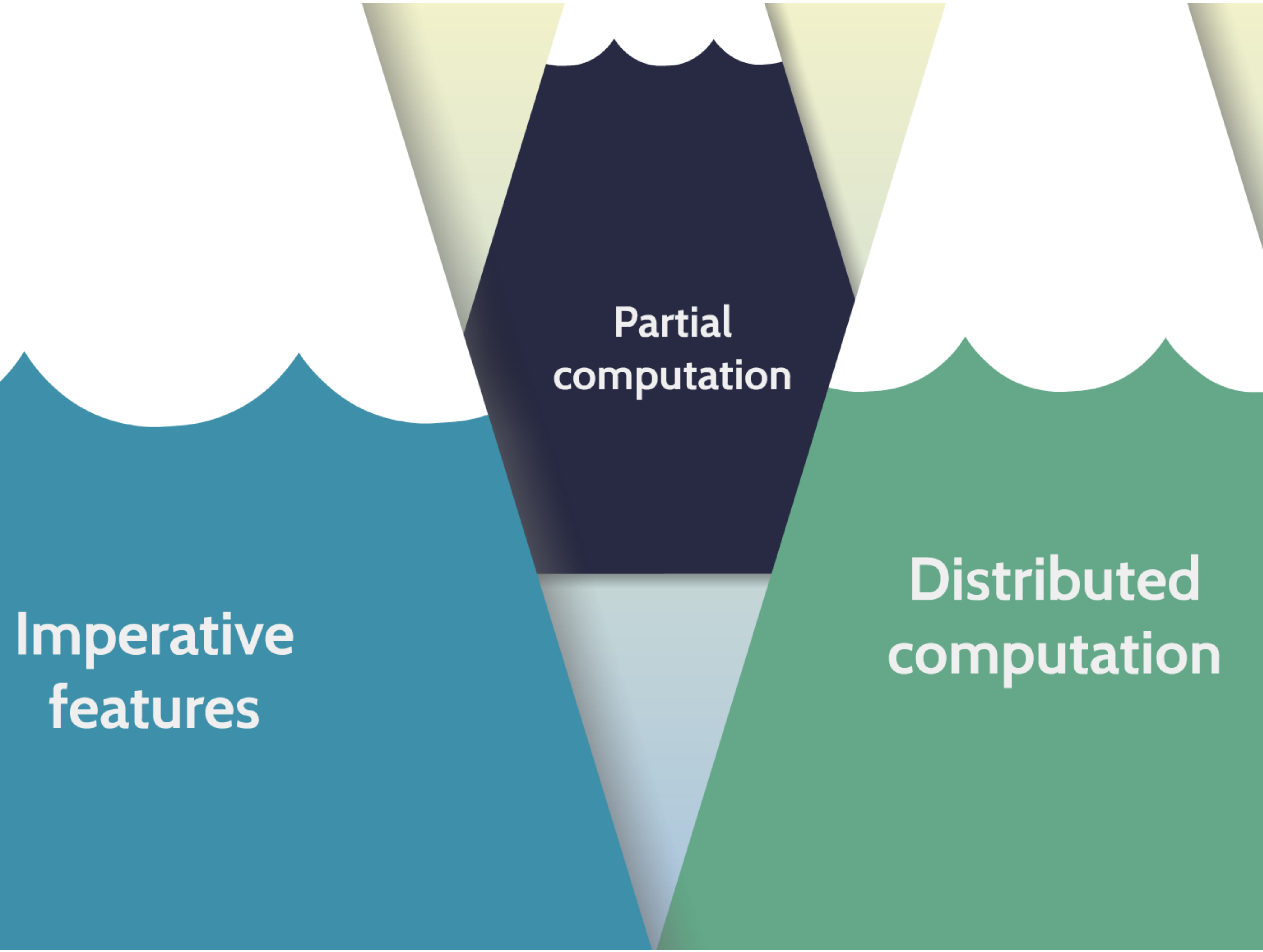
Program extraction

Type Theory based Proof Assistants (e.g. Coq) allow to extract programs from proofs

Curry-Howard-de Bruijn isomorphism

$$\frac{\text{proofs}}{\text{programs}} = \frac{\text{propositions}}{\text{types}} = \frac{\text{implementation}}{\text{specification}}$$

Pure Functional programs


Partial computation
Imperative features
Distributed computation

How to extract certified programs with effects?

Ad hoc type theories?

- Non-functional programming languages usually don't have a type theory supporting a Curry-Howard isomorphism
- Implementing ad hoc proof-assistants is intimidating!

Simulate effects in Coq?

- Encode (by deep embedding) of all effects within the proof assistant
- Cumbersome and inefficient code
- Difficult to reason about

?

Ad hoc type theories?

- Non-functional programming languages usually don't have a type theory supporting a Curry-Howard isomorphism
- Implementing ad hoc proof-assistants is intimidating!

Simulate effects in Coq?

- Encode (by deep embedding) of all effects within the proof assistant
- Cumbersome and inefficient code
- Difficult to reason about

HASKELL

Extend Coq with Monads

Extend Coq with Monads

HASKELL

Maybe Monad

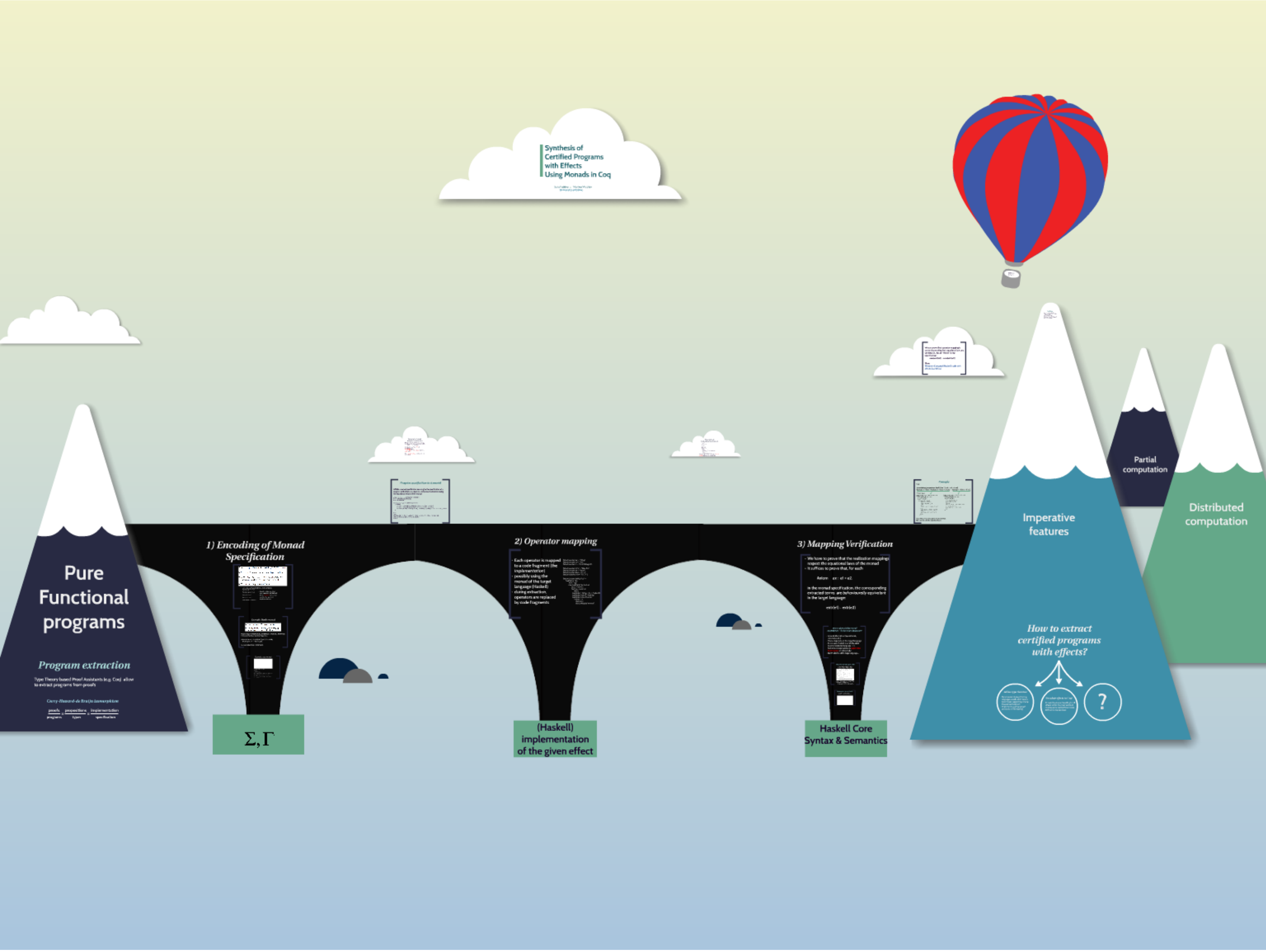
```
return :: a -> Maybe a
   return x  = Just x

   (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
   (>>=) m g = case m of
               Nothing -> Nothing
               Just x  -> g x
```

Store Monad

```
state :: (s -> (a, s)) -> State s a
return :: a -> State s a
return x = state ( \st -> (x, st) )

(>>=) :: State s a -> (a -> State s b) -> State s b
```



Synthesis of
Certified Programs
with Effects
Using Monads in Coq
Pure Functional programs
Program extraction
1) Encoding of Monad Specification
2) Operator mapping
3) Mapping Verification
Σ,Γ
(Haskell) implementation of the given effect
Haskell Core Syntax & Semantics
Imperative features
Partial computation
Distributed computation
How to extract certified programs with effects?
?

e
onal
ms

raction

nts (e.g. Coq) allow

omorphism

lementation

pecification

1) Encoding of Monad Specification

A monad specification is a triple (T, Σ, Γ) where

- $T : Set \to Set$ is the monadic type constructor;
- $\Sigma = \{op_i : \alpha_i \to TA_i\}$ is a set of operators (besides standard "return" and "bind");
- Γ is a set of equations of terms.

```
Module Type A_MONAD_INTERFACE <: MONAD_INTERFACE.
Parameter op1 : A1 -> M A.
...
Parameter opn : An -> M A.

Axiom eq1 : e1 = e1'.
...
Axiom eqn : en = en'.

End A_MONAD_INTERFACE.
```

- Inherits return and *bind* from MONAD_INTERFACE
- It's an interface: we do not provide the operators' implementations

Example: Maybe monad

- $\Sigma = \{nothing_A : TA, return, bind\}$
- $\Gamma = \{\forall f. bind(f, nothing_A) = nothing_B\}$+ the standard ones for bind and return

```
Module Type MAYBEMONAD_INTERFACE <: MONAD_INTERFACE.
Parameter Nothing : forall (A: Type), M A.

Axiom Strictness : forall (A B : Type) (f : A -> M B),
  (Nothing A) >>= f = (Nothing B).

End MAYBEMONAD_INTERFACE.
```

Example: state monad

Σ, Γ

1) Encoding of Monad Specification

A monad specification is a triple (T, Σ, Γ) where

- $T : Set \rightarrow Set$ is the monadic type constructor;
- $\Sigma = \{op_i : \alpha_i \rightarrow TA_i\}$ is a set of operators (besides standard "return" and "bind");
- Γ is a set of equations of terms.

```
Module Type A_MONAD_INTERFACE <: MONAD_INTERFACE.
Parameter op1 : A1 -> M A.
....
Parameter opn : A1 -> M A.

Axiom eq1 : e1 = e1'.
...
Axiom eqn : en = en'.

End A_MONAD_INTERFACE.
```

- Inherits *return* and *bind* from MONAD_INTERFACE
- It's an interface: we do not provide the operators' implementations

Example: Maybe monad

- $\Sigma = \{nothing_A : TA, return, bind\}$
- $\Gamma = \{\forall f. bind(f, nothing_A) = nothing_B\} +$

A monad specification is a triple (T, Σ, Γ) where

- $T : Set \to Set$ is the monadic type constructor;
- $\Sigma = \{op_i : \alpha_i \to T A_i\}$ is a set of operators (besides standard "return" and "bind");
- Γ is a set of equations of terms.

```
Module Type A_MONAD_INTERFACE <: MONAD_INTERFACE.
Parameter op1 : A1 -> M A.
....
Parameter opn : A1 -> M A.

Axiom eq1 : e1 = e1'.
...
Axiom eqn : en = en'.

End A_MONAD_INTERFACE.
```

- Inherits *return* and *bind* from MONAD_INTERFACE
- It's an interface: we do not provide the operators' implementations

(besides standard return and bind

- Γ is a set of equations of terms.

```
Module Type A_MONAD_INTERFACE <: MONAD_INTERFACE.
Parameter op1 : A1 -> M A.
....
Parameter opn : A1 -> M A.

Axiom eq1 : e1 = e1'.
...
Axiom eqn : en = en'.

End A_MONAD_INTERFACE.
```

- Inherits *return* and *bind* from MONAD_INTERFACE
- It's an interface: we do not provide the operators' implementations

Example: Maybe monad

- $\Sigma = \{nothing_A : TA, return, bind\}$
- $\Gamma = \{\forall f. bind(f, nothing_A) = nothing_B\} +$ the standard ones for bind and return

```
Module Type MAYBEMONAD_INTERFACE <: MONAD_INTERFACE.
Parameter Nothing : forall (A: Type), M A.

Axiom Strictness : forall (A B : Type) (f : A -> M B),
  (Nothing A) >>= f = (Nothing B).

End MAYBEMONAD_INTERFACE.
```

Example: state monad

$$\forall A : Set, \forall l : Loc, \forall x : GS(A), (\texttt{lookup}(l) \texttt{ >>= } (\lambda v.(\texttt{update}(l, v) \texttt{ >>= } x))) = x.$$

$$\forall A : Set, \forall l : Loc, \forall f : Value \to Value \to GS(A),$$
$$(\texttt{lookup}(l) \texttt{ >>= } (\lambda x.\texttt{lookup}(l) \texttt{ >>= } (\lambda y.(fxy)))) = (\texttt{lookup}(l) \texttt{ >>= } (\lambda x.(fxx))).$$

$$\forall A : Set, \forall l : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \to GS(A),$$
$$(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.(\texttt{update}(l, v') \texttt{ >>= } x))) = (\texttt{update}(l, v') \texttt{ >>= } x).$$

$$\forall A : Set, \forall l : Loc, \forall v, v' : Value, \forall f : Value \to GS(A),$$
$$(\texttt{update}(l, v) \texttt{ >>= } \lambda\_.(\texttt{lookup}(l) \texttt{ >>= } f)) = (\texttt{update}(l, v) \texttt{ >>= } \lambda\_.fv).$$

$$\forall A : Set, \forall l, l' : Loc, \forall f : Value \to Value \to GS(A), l \neq l' \to$$
$$(\texttt{lookup}(l) \texttt{ >>= } (\lambda v.(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.(fvv'))))) =$$
$$(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.(\texttt{lookup}(l) \texttt{ >>= } \lambda v.(fvv')))).$$

$$\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \to GS(A), l \neq l' \to$$
$$(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.(\texttt{update}(l', v') \texttt{ >>= } x))) =$$
$$(\texttt{update}(l', v') \texttt{ >>= } (\lambda\_.(\texttt{update}(l, v) \texttt{ >>= } x))).$$

$$\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall f : Value \to GS(A), l \neq l' \to$$
$$(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.f\ v')))) =$$
$$(\texttt{lookup}(l') \texttt{ >>= } (\lambda v'.(\texttt{update}(l, v) \texttt{ >>= } (\lambda\_.f\ v')))).$$

```
Module Type STATEMONAD_INTERFACE <: MONAD_INTERFACE.
Include MONAD_INTERFACE.
Parameter loc : Set.
Parameter val: Set.
Parameter st :Set.
Parameter lookUp: forall (A: loc), M val.
Parameter update: forall (A: loc) (a :val), M unit.

Axiom lookUp_idempotence:
forall (l : loc ) (f : val-> val-> M val),  (lookUp l) >>= (fun x =>(lookUp l)>>= (fun y => (f x y))) = lookUp l >>= (fun x => (f x x)).

Axiom update_idempotence:
forall (v v' : val) (l : loc) (x : unit -> M val), (update l v) >>= (fun _ => (update l v') >>= x) = (update l v') >>= x.

Axiom lookUp_after_update :
forall (v : val) (l : loc) (f : val -> M val), (update l v) >>= (fun _ => (lookUp l >>= f)) = (update l v) >>= (fun _ => f v).
....
End STATEMONAD_INTERFACE.
```

Example: state monad

$\forall A : Set, \forall l : Loc, \forall x : GS(A), (\texttt{lookup}(l) \;\texttt{>>=}\; (\lambda v.(\texttt{update}(l,v) \;\texttt{>>=}\; x))) = x.$

$\forall A : Set, \forall l : Loc, \forall f : Value \to Value \to GS(A),$
$(\texttt{lookup}(l) \;\texttt{>>=}\; (\lambda x.\texttt{lookup}(l) \;\texttt{>>=}\; (\lambda y.(f x y)))) = (\texttt{lookup}(l) \;\texttt{>>=}\; (\lambda x.(f x x))).$

$\forall A : Set, \forall l : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \to GS(A),$
$(\texttt{update}(l,v) \;\texttt{>>=}\; (\lambda\_.(\texttt{update}(l,v') \;\texttt{>>=}\; x))) = (\texttt{update}(l,v') \;\texttt{>>=}\; x).$

$\forall A : Set, \forall l : Loc, \forall v, v' : Value, \forall f : Value \to GS(A),$
$(\texttt{update}(l,v) \;\texttt{>>=}\; \lambda\_.(\texttt{lookup}(l) \;\texttt{>>=}\; f)) = (\texttt{update}(l,v) \;\texttt{>>=}\; \lambda\_.f v).$

$\forall A : Set, \forall l, l' : Loc, \forall f : Value \to Value \to GS(A), l \neq l' \to$
$(\texttt{lookup}(l) \;\texttt{>>=}\; (\lambda v.(\texttt{lookup}(l') \;\texttt{>>=}\; (\lambda v'.(f v v'))))) =$
$(\texttt{lookup}(l') \;\texttt{>>=}\; (\lambda v'.(\texttt{lookup}(l) \;\texttt{>>=}\; \lambda v.(f v v')))).$

$\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \to GS(A), l \neq l' \to$
$(\texttt{update}(l,v) \;\texttt{>>=}\; (\lambda\_.(\texttt{update}(l',v') \;\texttt{>>=}\; x))) =$
$(\texttt{update}(l',v') \;\texttt{>>=}\; (\lambda\_.(\texttt{update}(l,v) \;\texttt{>>=}\; x))).$

$\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall f : Value \to GS(A), l \neq l' \to$
$(\texttt{update}(l,v) \;\texttt{>>=}\; (\lambda\_.(\texttt{lookup}(l') \;\texttt{>>=}\; (\lambda v'.f\ v')))) =$
$(\texttt{lookup}(l') \;\texttt{>>=}\; (\lambda v'.(\texttt{update}(l,v) \;\texttt{>>=}\; (\lambda\_.f\ v')))).$

Module Type STATEMONAD_INTERFACE <: MONAD_INTERFACE.
Include MONAD_INTERFACE.
Parameter loc : Set.

$$\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall x : \texttt{unit} \rightarrow GS(A), l \neq l' \rightarrow$$
$$(\texttt{update}(l, v)\ \texttt{>>=}\ (\lambda\_.(\texttt{update}(l', v')\ \texttt{>>=}\ x))) = (\texttt{update}(l', v')\ \texttt{>>=}\ (\lambda\_.(\texttt{update}(l, v)\ \texttt{>>=}\ x))).$$
$$\forall A : Set, \forall l, l' : Loc, \forall v, v' : Value, \forall f : Value \rightarrow GS(A), l \neq l' \rightarrow$$
$$(\texttt{update}(l, v)\ \texttt{>>=}\ (\lambda\_.(\texttt{lookup}(l')\ \texttt{>>=}\ (\lambda v'.f\ v')))) = (\texttt{lookup}(l')\ \texttt{>>=}\ (\lambda v'.(\texttt{update}(l, v)\ \texttt{>>=}\ (\lambda\_.f\ v')))).$$

```
Module Type STATEMONAD_INTERFACE <: MONAD_INTERFACE.
Include MONAD_INTERFACE.
Parameter loc : Set.
Parameter val: Set.
Parameter st :Set.
Parameter lookUp: forall (A: loc), M val.
Parameter update: forall (A: loc) (a :val), M unit.

Axiom lookUp_idempotence:
forall (l : loc ) (f : val-> val-> M val),  (lookUp l) >>= (fun x =>(lookUp l)>>= (fun y => (f x y))) = lookUp l >>= (fun x => (f x x)).

Axiom update_idempotence:
forall (v v' : val) (l : loc) (x : unit -> M val), (update l v) >>= (fun _ => (update l v') >>= x) = (update l v') >>= x.

Axiom lookUp_after_update :
forall (v : val) (l : loc) (f : val -> M val), (update l v) >>= (fun _ => (lookUp l >>= f)) = (update l v) >>= (fun _ => f v).
....
End STATEMONAD_INTERFACE.
```

Program specification in a monad

Within a monad specification we can give the specification of a program with effects as a Lemma, and prove its existence using the equational theory of the monad

```
Module StateInstance <: STATEMONAD_INTERFACE.
Include STATEMONAD_INTERFACE.
Include MemoryState.

Lemma swap_program : forall (l1 l2 : loc), l1 <> l2 ->
    {c : M unit |
        ((c >>= (fun _ => lookUp l2)) = (lookUp l1) >>= fun x => c >>= (fun _ => ret x)) /\
        ((c >>= (fun _ => lookUp l1)) = (lookUp l2) >>= fun x => c >>= (fun _ => ret x)) /\
        forall (l : loc), (l <> l1 /\ l <> l2) ->((c >>= fun _ => lookUp(l))) = ((lookUp(l) >>= fun x => c >>= fun _ => ret x ))
    }.

Proof.
intros.
exists ((lookUp l1)>>= (fun x => lookUp l2 >>= (fun y => (update l1 y) >>= (fun _ => update l2 x)))).
... (here we Rewrite using the axioms of the monad) ...
Defined.
```

Extracted code (with undefined constructors)

From a Coq term of monadic type we can obtain Haskell programs using standard Extraction facility:

t:(M A) ---> extr(t) :: (M A)

```
swap_program :: Loc -> Loc -> (M Unit)
swap_program l1 l2 =
  bind (lookUp l1) (\x ->
   bind (lookUp l2) (\y -> bind (update l1 y) (\x0 ->
update l2 x)))
```

Notice: monadic operators are not defined (have still to be realized).

Program specification in a monad

Within a monad specification we can give the specification of a program with effects as a Lemms, and prove its existence using the equational theory of the monad

```
Module StateInstance <: STATEMONAD_INTERFACE.
Include STATEMONAD_INTERFACE.
Include MemoryState.

Proof.
intros.
Defined.
```

2) Operator mapping

- Each operator is mapped to a code fragment (the *implementation*)
- possibly using the monad of the target language (Haskell)
- during extraction, operators are replaced by code fragments

```
Extract Constant loc => "String".
Extract Constant val => "Int".
Extract Constant st => "([]) ((,) String Int)".

Extract Constant M "a" => "State St a".
Extract Constant ret => "return".
Extract Constant bind => "(>>=)".
Extract Inductive unit => "()" [ "()" ].

Extract Constant lookUp "loc" =>
    "lookUp loc = do
        mem <- get
        case LookUpList' loc mem of
            Just s -> return s
            Nothing -> return 0
              where
            LookUpList' :: String -> St -> Maybe Val
            LookUpList' name [] = Nothing
            LookUpList' name ((n,v):xs) =
              if name == n
                then Just v
                else LookUpList' name xs".
```

(Haskell) implementation of the given effect

2) Operator mapping

- Each operator is mapped to a code fragment (the *implementation*)
- possibly using the monad of the target language (Haskell)
- during extraction, operators are replaced by code fragments

```
Extract Constant loc => "String".
Extract Constant val => "Int".
Extract Constant st => "([]) ((,) String Int)".

Extract Constant M "a"=> "State St a".
Extract Constant ret => "return".
Extract Constant bind => "(>>=)".
Extract Inductive unit => "()" [ "()" ].

Extract Constant lookUp "loc" =>
    "lookUp loc = do
        mem <- get
        case LookUpList' loc mem of
            Just s -> return s
            Nothing -> return 0
               where
             LookUpList' :: String -> St -> Maybe Val
             LookUpList' name [] = Nothing
             LookUpList' name ((n,v):xs) =
                if name == n
                  then Just v
                  else LookUpList' name xs".
```

Extracted code (with defined constructors)

```
type M a = State St a
ret :: a1 -> M a1
ret = return
bind :: (M a1) -> (a1 -> M a2) -> M a2
bind = (>>=)

type Loc = String
type Val = Int
type St = ([]) ((,) String Int)

lookUp :: Loc -> M Val
lookUp = lookUp loc = do
  mem <- get
  case varLookUpList' loc mem of
   Just s -> return s
   Nothing -> return 0
   where
    varLookUpList' :: String -> St -> Maybe Val
    varLookUpList' name [] = Nothing
    varLookUpList' name ((n,v):xs) = if name == n then Just v else varLookUpList' name xs

swap_program :: Loc -> Loc -> (M ())
swap_program l1 l2 =
  bind (lookUp l1) (\x ->
    bind (lookUp l2) (\y -> bind (update l1 y) (\x0 -> update l2 x)))
```

Now monadic operators are fully defined but not certified (code fragment can be anything)

Example

Coq:

Axiom lookup_idempotence: forall (l : loc) (f : val -> val -> M val),
(lookUp l) >>= (fun x ->(lookUp l)>>= (fun y -> (f x y))) = lookUp l >>= (fun x => (f x x)).

Haskell Core:

These two terms can be proved to be bisimilar.
(The same for all other equational laws)

3) Mapping Verification

- We have to prove that the realization mappings respect the equational laws of the monad
- It suffices to prove that, for each

  Axiom ax : e1 = e2.

  in the monad specification, the corresponding extracted terms are *behaviourally equivalent* in the target language:

  extr(e1) ~ extr(e2)

How to define behavioural equivalence – in the target language?

- Several alternatives (operational, denotational...)
- Choice depends on the target language
- In our case, Haskell is a call-by-need purely functional language ==> behavioural equivalence is applicative bisimulation a la Abramsky
- But Haskell is still a huge language...

Haskell Core Syntax & Semantics

Imper
feat

How
certifie
with

Ad hoc type theories?

- Non-functional programming languages usually don't have a type theory supporting a Curry-Howard isomorphism
- Implementing ad hoc proof-assistants is intimidating!

3) Mapping Verification

- We have to prove that the realization mappings respect the equational laws of the monad
- It suffices to prove that, for each

  Axiom ax : e1 = e2.

  in the monad specification, the corresponding extracted terms are *behaviourally equivalent* in the target language:

  extr(e1) ~ extr(e2)

et language:

extr(e1) ~ extr(e2)

How to define behavioural equivalence ~ in the target language?

- Several alternatives (operational, denotational...)
- Choice depends on the target language
- In our case, Haskell is a call-by-need purely functional language ==> behavioural equivalence is applicative bisimulation a la Abramsky
- But Haskell is still a huge language...

Haskell Core (aka System FC)

- in ghc, Haskell compiles to Core, a variant of System F with coercions

$$E ::= V \mid (E\ E) \mid \lambda V.E \mid (cast\ E\ C) \mid (letrec\ V_1 = E_1,\ \ldots,\ V_n = E_n\ in\ E)$$
$$\mid (c_i E_1 \ldots E_{ar(c_i)}) \mid (case_k\ E\ of\ Alt_1\ \ldots\ Alt_{|D_K|})$$
$$Alt_i = ((c_i V_1 \ldots V_{ar(c_i)}) \rightarrow E)$$

$$\text{VAR}\ \frac{\Sigma(n) = e}{\Sigma \vdash_{op} n \rightarrow e} \qquad \text{BETA}\ \frac{}{\Sigma \vdash_{op} (\lambda n.e_1)\ e_2 \rightarrow e_1[n \mapsto e_2]}$$

$$\text{APP}\ \frac{\Sigma \vdash_{op} e_1 \rightarrow e_1'}{\Sigma \vdash_{op} e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \text{LETNONREC}\ \frac{}{\Sigma \vdash_{op} \text{let } n = e_1 \text{ in } e_2 \rightarrow e_2[n \mapsto e_1]}$$

$$\text{LETREC}\ \frac{\Sigma, \overline{[n_i \mapsto e_i]}^i \vdash_{op} u \rightarrow u'}{\Sigma \vdash_{op} \text{let rec } \overline{n_i = e_i}^i \text{ in } u \rightarrow \text{let rec } \overline{n_i = e_i}^i \text{ in } u'}$$

$$\text{LETREC RETURN}\ \frac{fv(u) \cap \overline{n_i}^i = \emptyset}{\Sigma \vdash_{op} \text{let rec } \overline{n_i = e_i}^i \text{ in } u \rightarrow u}$$

$$\text{CASE}\ \frac{\Sigma \vdash_{op} e \rightarrow e'}{\Sigma \vdash_{op} \text{case } e \text{ as } n \text{ return } \tau \text{ of } \overline{\text{alt}_i}^i \rightarrow \text{case } e' \text{ as } n \text{ return } \tau \text{ of } \overline{\text{alt}_i}^i}$$

(and rules for pattern matching)

- given a Haskell program M, let core(M) be its translation in System FC
  - implemented by "ghc -ddump-simpl"

Equivalence for Core (and Haskell)

- Core is all we need for defining equivalence in the target language!

$\sim$ is the largest relation such that, for all s, t closed expressions, if $s \sim t$ then

- $\forall\, v,\ s \Downarrow v \ \Rightarrow\ \exists\, w$ such that $t \Downarrow w$, $(v\,\Omega) \sim (w\,\Omega)$ and $\forall\, letrec,\ case$ free r : $(v\,r) \sim (w\,r)$
- $\forall\, w,\ t \Downarrow w \ \Rightarrow\ \exists\, v$ such that $s \Downarrow v$, $(v\,\Omega) \sim (w\,\Omega)$ and $\forall\, letrec,\ case$ free r : $(v\,r) \sim (w\,r)$

Prop. In Core $\sim$ corresponds to contextual equivalence.

$\sim$ is lifted to Haskell programs as

$$P \sim_h Q \ \stackrel{\triangle}{\Longleftrightarrow}\ core(P) \sim core(Q)$$

Example

Coq:

Axiom lookup_idempotence: forall (l : loc) (f : val -> val -> M val),
(lookUp l) >>= (fun x ->(lookUp l)>>= (fun y -> (f x y))) = lookUp l >>= (fun x => (f x x)).

Haskell Core:

```
lookup_idempotence_Left :: Loc -> (Val -> Val -> M Val) -> M Val
lookup_idempotence_Left =
 \ (l_amS :: Loc) (f_amT :: Val -> Val -> M Val) ->
  (\ (eta_B1 :: [(String, Int)]) ->
   ((\ (eta_Xsv :: [(String, Int)]) ->
     let {
       a_ssq :: Identity (Val, [(String, Int)])
             a_ssq = a_sre l_amS eta_Xsv } in
            let {
             a_Xt8 :: Identity (Val, [(String, Int)])
             a_Xt8 =
              a_sre
               l_amS (case a_ssq `cast` ... of _ { (_, s'_asf) -> s'_asf
                 }) } in
            ((f_amT
              (case a_ssq `cast` ... of _ { (a5_as9, _) -> a5_as9 })
              (case a_Xt8 `cast` ... of _ { (a5_as9, _) -> a5_as9 }))
             `cast` ...)
             (case a_Xt8 `cast` ... of _ { (_, s'_asf) -> s'_asf }))
          `cast` ...)
          eta_B1)
        `cast` ...
```

```
lookup_idempotence_Right :: Loc -> (Val -> Val -> M Val) -> M Val
lookup_idempotence_Right =
 \ (l_amW :: Loc) (f_amX :: Val -> Val -> M Val) ->
  (\ (eta_B1 :: [(String, Int)]) ->
   ((\ (eta_Xsv :: [(String, Int)]) ->
     let {
      a_ssq :: Identity (Val, [(String, Int)])
      a_ssq = a_sre l_amW eta_Xsv } in
     let {
      x_amY :: Val
      x_amY = case a_ssq `cast` ... of _ { (a5_as9, _) -> a5_as9 }
        } in
     ((f_amX x_amY x_amY) `cast` ...)
      (case a_ssq `cast` ... of _ { (_, s'_asf) -> s'_asf }))
   `cast` ...)
    eta_B1)
  `cast` ...
```

These two terms can be proved to be bisimilar.
(The same for all other equational laws)

We can prove that operator mapping is correct by proving that equational laws are verified, i.e., for all "e1=e2" in the specification:

core(extr(e1)) ~ core(extr(e2))

Then:

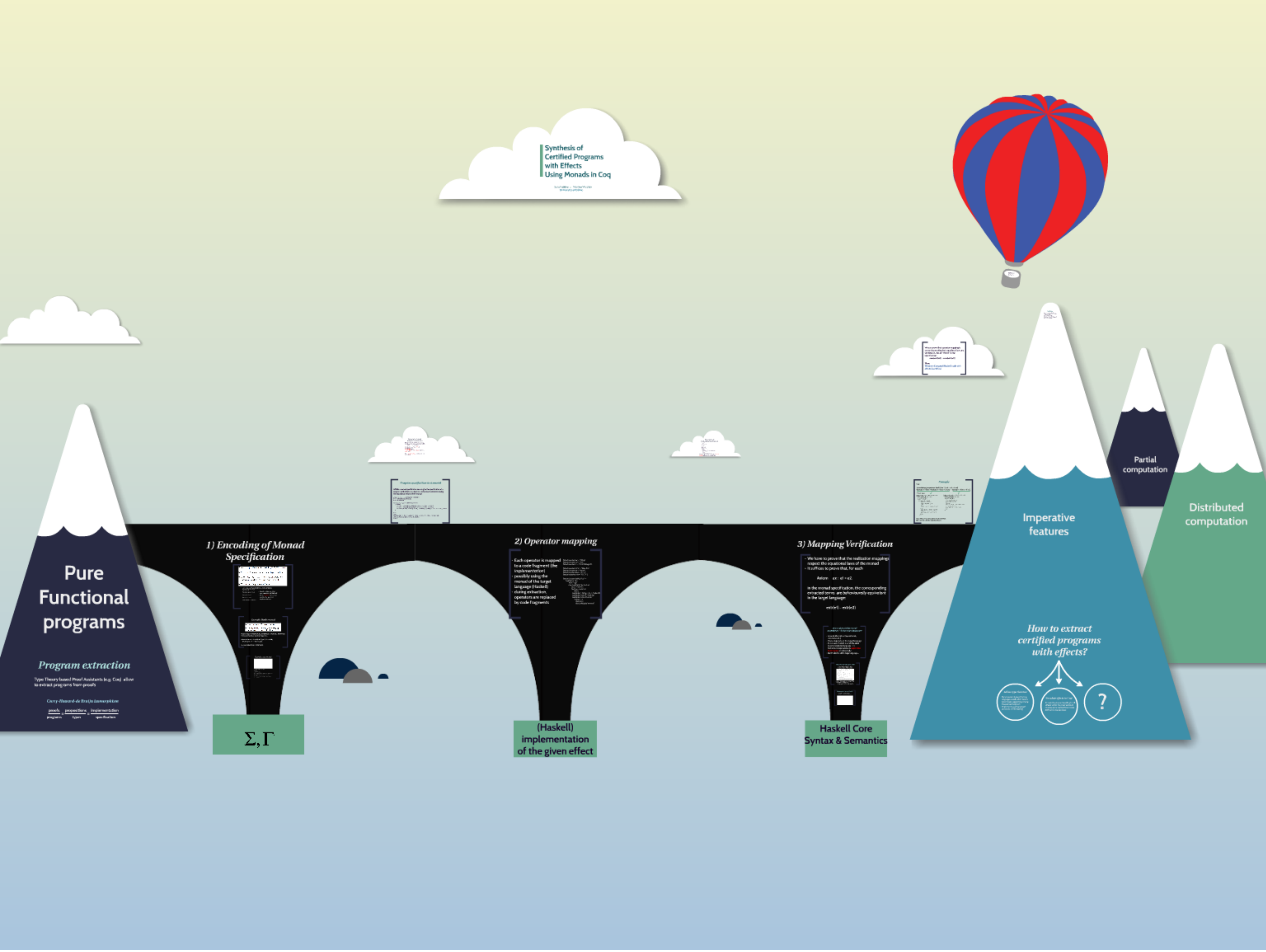
Theorem: Extracted (Haskell) code with effects is certified.

Conclusions

- Presented a general methodology for extracting certified programs with effects from proofs in Coq
- reuses existing technologies
- relies on monadic specification of effects
- correctness = preservation of equational laws in the extracted code

To do

- Formalize System FC semantics and equivalence proofs for some simple monad
- Automation of equational reasoning (e.g. deduction modulo? rewriting?)
- Derive logics (ad hoc for each monad) from equational theory, easier to use in specifications and proofs
    - e.g. for state monad: Hoare logics



Synthesis of Certified Programs with Effects Using Monads in Coq
Pure Functional programs
Program extraction
Type Theory based Proof Assistants (e.g. Coq) allow to extract programs from proofs
1) Encoding of Monad Specification
Σ,Γ
2) Operator mapping
(Haskell) implementation of the given effect
3) Mapping Verification
Haskell Core Syntax & Semantics
Imperative features
Partial computation
Distributed computation
How to extract certified programs with effects?
?