Synthesis of distributed mobile programs using monadic types in Coq

Marino Miculan   Marco Paviotti

Dept. of Mathematics and Computer Science
University of Udine

ITP 2012
August 13th, 2012

The problem

The extraction of certified *functional* and effect-free programs is a well-know practice in the field of Type Theory, however:

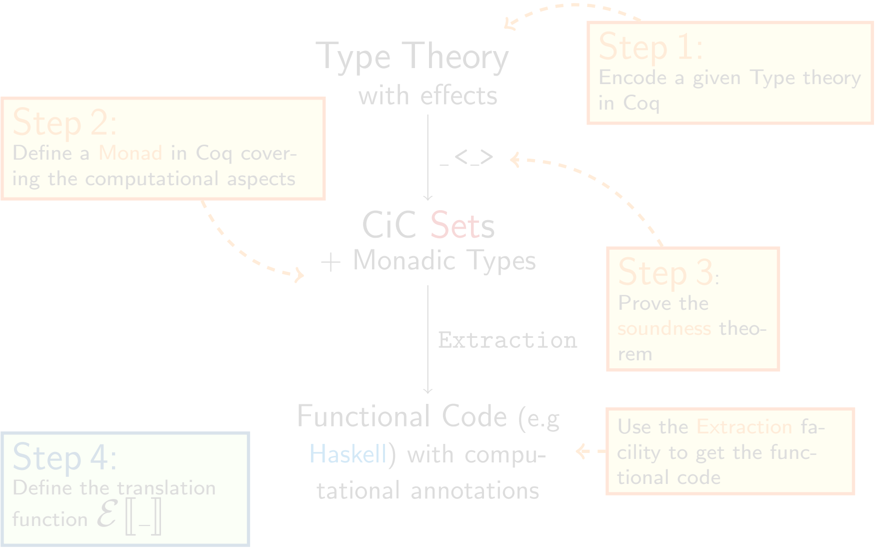
- ✔ There are many other computational effects (and corresponding Type Theories, possibly)
- ✔ These scenarios would greatly benefit from a mechanisms for extraction
- ✘ Languages implementing these aspects usually do not support the Curry-Howard isomorphism
- ✘ Implementing a specific proof-assistant would be a daunting task anyway.

Our contribution

We propose:

- a general methodology for circumventing this problem using the existing technology (Coq)
  - encapsulate non-functional aspects in monadic types
  - implement a post-extraction compiler for realizing monadic constructors in the target language
- example: distributed programs with effects in Erlang.

A general methodology



Type Theory
with effects

Step 1:
Encode a given Type theory in Coq

_<_>

Step 2:
Define a Monad in Coq covering the computational aspects

CiC Sets
+ Monadic Types

Step 3:
Prove the soundness theorem

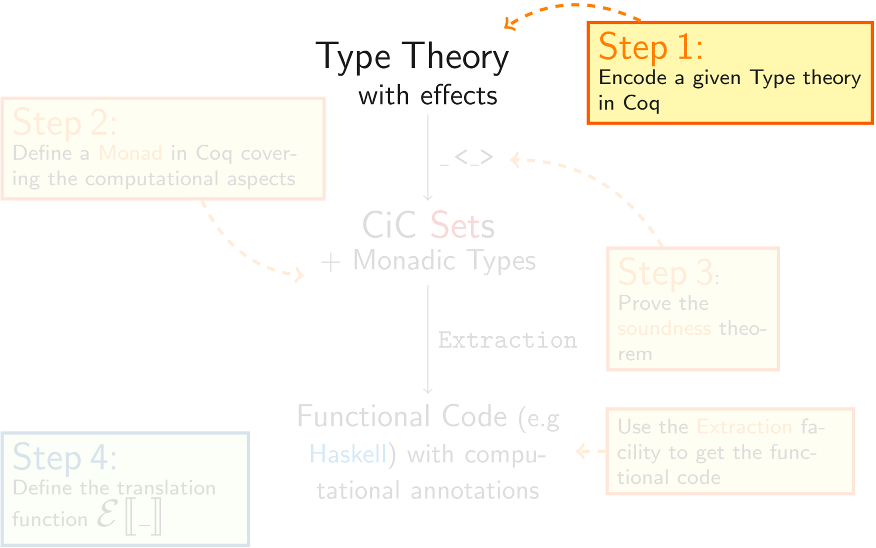
`Extraction`

Functional Code (e.g Haskell) with computational annotations

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

_<_>

CiC Sets
+ Monadic Types

Extraction

Functional Code (e.g Haskell) with computational annotations

Step 1:
Encode a given Type theory in Coq

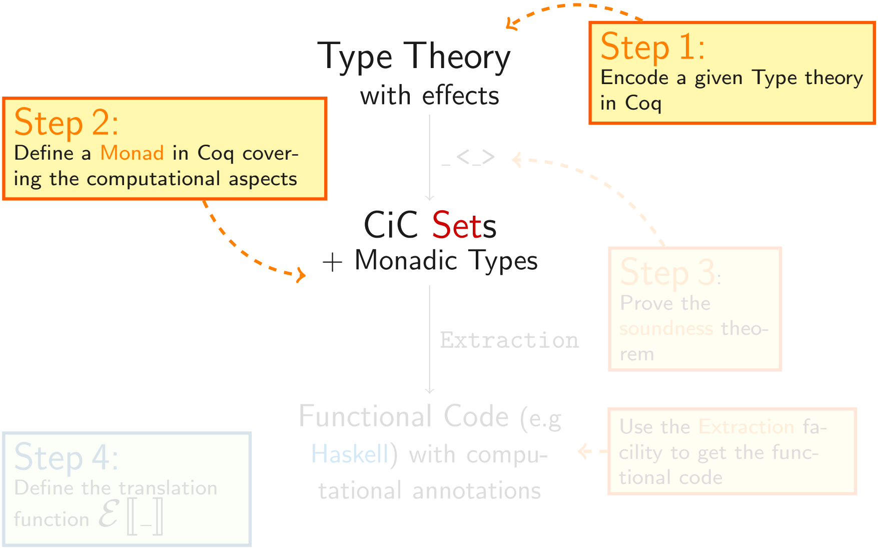
Step 2:
Define a Monad in Coq covering the computational aspects

Step 3:
Prove the soundness theorem

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

_<_>

CiC **Sets**
+ Monadic Types

`Extraction`

Functional Code (e.g Haskell) with computational annotations

Step 1:
Encode a given Type theory in Coq

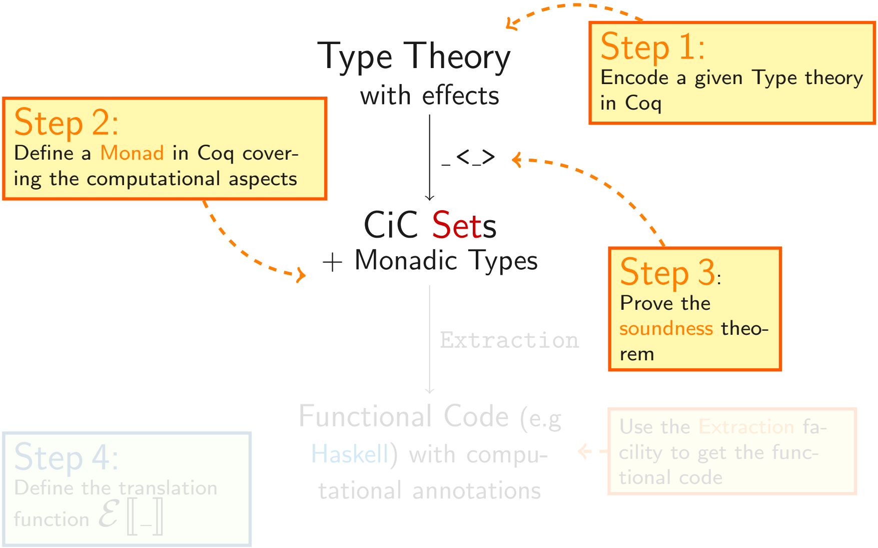
Step 2:
Define a Monad in Coq covering the computational aspects

Step 3:
Prove the soundness theorem

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

Step 1:
Encode a given Type theory in Coq

Step 2:
Define a Monad in Coq covering the computational aspects

_<_>

CiC Sets
+ Monadic Types

Step 3:
Prove the soundness theorem

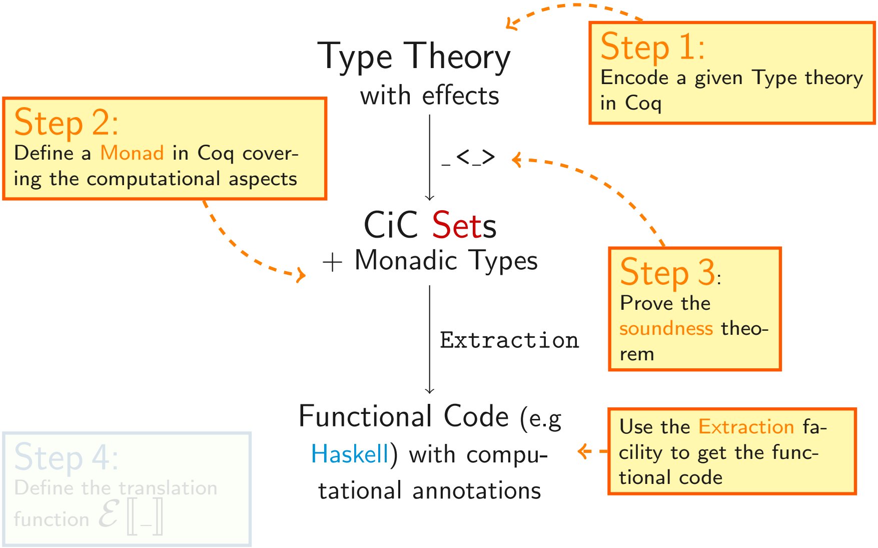
`Extraction`

Functional Code (e.g Haskell) with computational annotations

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

Step 1:
Encode a given Type theory in Coq

Step 2:
Define a Monad in Coq covering the computational aspects

`_<_>`

CiC Sets
+ Monadic Types

Step 3:
Prove the soundness theorem

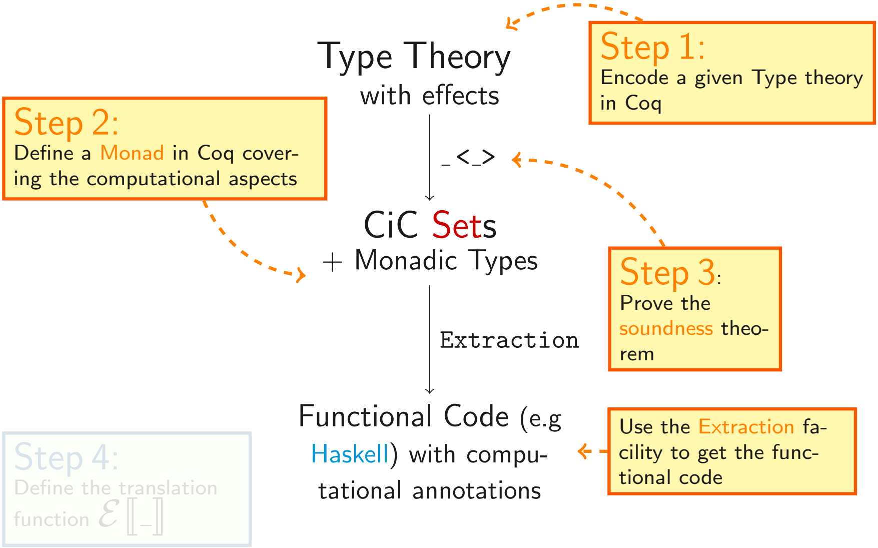
`Extraction`

Functional Code (e.g Haskell) with computational annotations

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

`_<_>`

CiC Sets
$+$ Monadic Types

`Extraction`

Functional Code (e.g Haskell) with computational annotations

Step 1:
Encode a given Type theory in Coq

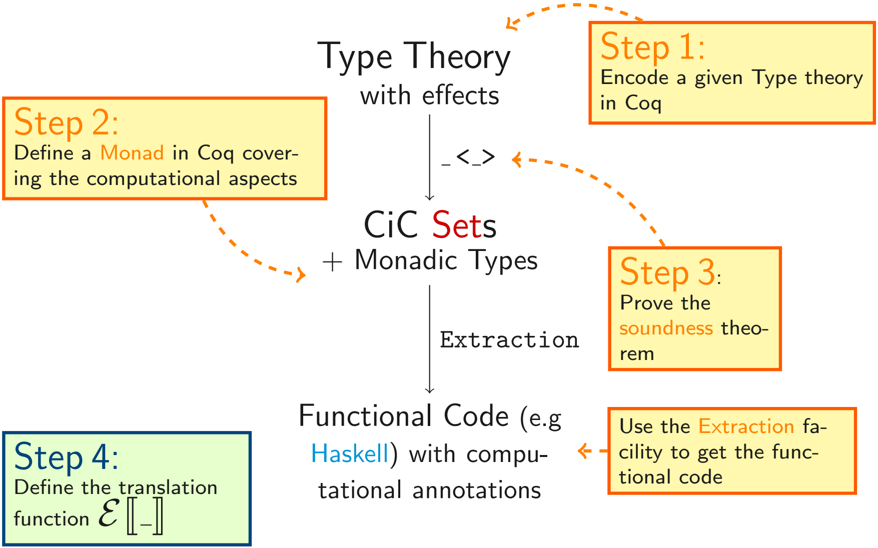
Step 2:
Define a Monad in Coq covering the computational aspects

Step 3:
Prove the soundness theorem

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

Step 1:
Encode a given Type theory in Coq

Step 2:
Define a Monad in Coq covering the computational aspects

_<_>

CiC Sets
+ Monadic Types

Step 3:
Prove the soundness theorem

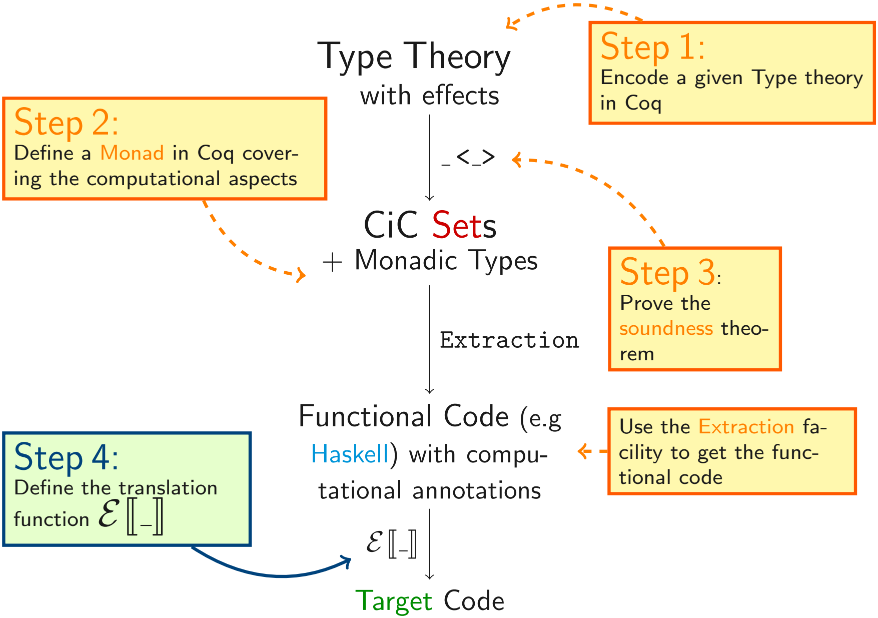
Extraction

Functional Code (e.g Haskell) with computational annotations

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

`_<_>`

CiC Sets
+ Monadic Types

`Extraction`

Functional Code (e.g Haskell) with computational annotations

$\mathcal{E}[\![\_]\!]$

Target Code

Step 1:
Encode a given Type theory in Coq

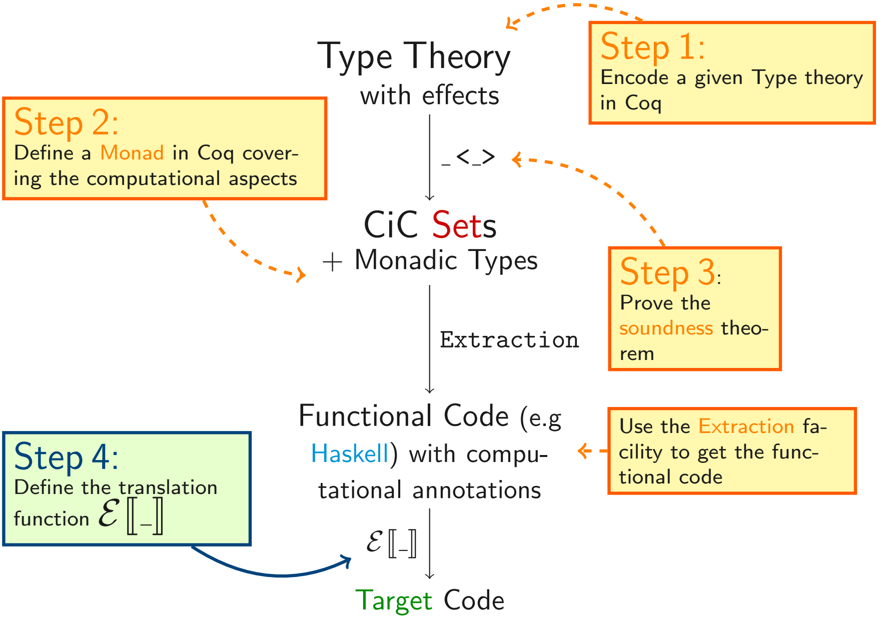
Step 2:
Define a Monad in Coq covering the computational aspects

Step 3:
Prove the soundness theorem

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

`_<_>`

CiC Sets
+ Monadic Types

`Extraction`

Functional Code (e.g Haskell) with computational annotations

$\mathcal{E}[\![\_]\!]$

Target Code

Step 1:
Encode a given Type theory in Coq

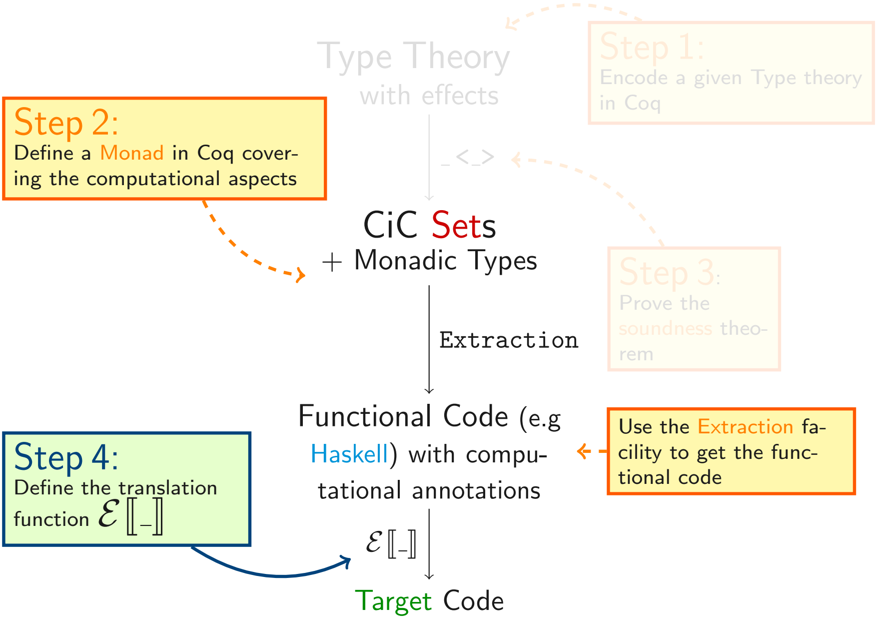
Step 2:
Define a Monad in Coq covering the computational aspects

Step 3:
Prove the soundness theorem

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

A general methodology



Type Theory
with effects

Step 1:
Encode a given Type theory in Coq

_<_>

Step 2:
Define a Monad in Coq covering the computational aspects

CiC Sets
+ Monadic Types

Step 3:
Prove the soundness theorem

`Extraction`

Functional Code (e.g Haskell) with computational annotations

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

$\mathcal{E}[\![\_]\!]$

Target Code

Extraction of distributed code

We define the distributed monad in the **Calculus of Inductive Constructions**

`forall w: World` and `A: Set`

`IO w A :Set`

Extraction of distributed code

We define the distributed monad in the **Calculus of Inductive Constructions**

`forall w: World` and `A: Set`

`IO w A :Set`

A computation localized on the specified host

Extraction of distributed code

We define the distributed monad in the **Calculus of Inductive Constructions**

$$\overset{\texttt{forall w: World} \text{ and } \texttt{A: Set}}{\texttt{IO w A :Set}}$$

By Curry-Howard Isomorphism, the (constructive) proofs of these specifications are turned into **decorated Haskell** code

$$\texttt{IO w A} \overset{\textit{Extraction}}{\Longrightarrow} \mathbb{H}$$

Extraction of distributed code

We define the distributed monad in the **Calculus of Inductive Constructions**

`forall w: World` and `A: Set`

$$\texttt{IO w A} : \texttt{Set}$$

By Curry-Howard Isomorphism, the (constructive) proofs of these specifications are turned into **decorated Haskell** code

$$\texttt{IO w A} \overset{\textit{Extraction}}{\Rightarrow} \mathbb{H}$$

These decorations are exploited by the Haskell-**Erlang** Compiler

$$\mathcal{E}[\![\_]\!] : \mathbb{H} \rightarrow \mathbb{E}$$

Extraction of distributed code

We define the distributed monad in the **Calculus of Inductive Constructions**

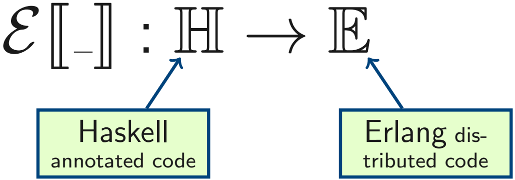
`forall w: World` and `A: Set`

$$\texttt{IO w A :Set}$$

By Curry-Howard Isomorphism, the (constructive) proofs of these specifications are turned into **decorated Haskell** code

$$\texttt{IO w A} \overset{\textit{Extraction}}{\Longrightarrow} \mathbb{H}$$

These decorations are exploited by the Haskell-**Erlang** Compiler



$$\mathcal{E}[\![\_]\!] : \mathbb{H} \longrightarrow \mathbb{E}$$

Haskell
annotated code

Erlang distributed code

Monads in Coq

We define a *family* of monads indexed by worlds from `Set` to `Set`. Given a world `w` a monad is a functor defined as

$$\mathtt{IO\ w\ A} = S \ \rightarrow ((\mathtt{R\ w\ A}) + \mathtt{Error})$$

Monads in Coq

We define a *family* of monads indexed by worlds from `Set` to `Set`. Given a world `w` a monad is a functor defined as

$$\mathtt{IO}\ \mathtt{w}\ \mathtt{A} = S \ \rightarrow ((\mathtt{R}\ \mathtt{w}\ \mathtt{A}) + \mathtt{Error})$$

A **Localized** computation

Monads in Coq

We define a *family* of monads indexed by worlds from `Set` to `Set`. Given a world `w` a monad is a functor defined as

$$\mathtt{IO\ w\ A} = S \rightarrow ((\mathtt{R\ w\ A}) + \mathtt{Error})$$

Function Space
from the global store to the results plus the error state

Monads in Coq

We define a *family* of monads indexed by worlds from `Set` to `Set`. Given a world `w` a monad is a functor defined as

$$\texttt{IO w A} = S \rightarrow ((\texttt{R w A}) + \texttt{Error})$$

Monadic Operators

$$\texttt{IOget}_{\texttt{w}}\ \texttt{A} : \texttt{IO remote A} \rightarrow \texttt{IO w A}$$

$\lambda\ \kappa\ \sigma.\ \kappa(\sigma)$ (Operator's implementation)

Monads in Coq

We define a *family* of monads indexed by worlds from `Set` to `Set`. Given a world `w` a monad is a functor defined as

$$\mathtt{IO\ w\ A} = S \rightarrow ((\mathtt{R\ w\ A}) + \mathtt{Error})$$

Monadic Operators

$$\mathtt{IOget_w\ A} : \mathtt{IO\ remote\ A} \rightarrow \mathtt{IO\ w\ A}$$

$$\lambda\ \kappa\ \sigma.\ \kappa(\sigma)$$ (Operator's implementation)

Other monadic operators

$\mathtt{IOreturn_w\ A} : A \rightarrow \mathtt{IO\ w\ A}$

$\mathtt{IObind_w\ A\ B} : \mathtt{IO\ w\ A} \rightarrow (A \rightarrow \mathtt{IO\ w\ B}) \rightarrow \mathtt{IO\ w\ B}$

$\mathtt{IOlookup_w\ A} : \mathtt{Ref_w} \rightarrow (\mathbb{N} \rightarrow \mathtt{IO\ w\ A}) \rightarrow \mathtt{IO\ w\ A}$

$\mathtt{IOupdate_w\ A} : \mathtt{Ref_w} \rightarrow \mathbb{N} \rightarrow \mathtt{IO\ w\ A} \rightarrow \mathtt{IO\ w\ A}$

$\mathtt{IOnew_w\ A} : \mathbb{N} \rightarrow (\mathtt{Ref_w} \rightarrow \mathtt{IO\ w\ A}) \rightarrow \mathtt{IO\ w\ A}$

Extraction

Lemma (Remote Procedure Call)

$\forall \mathtt{w}\ \mathtt{w'}, (\mathbb{N} \to \mathtt{IO}\ \mathtt{w'}\ \mathtt{bool}) \to (\mathbb{N} \to \mathtt{IO}\ \mathtt{w}\ \mathtt{bool})$

Proof. simpl; introv f. intro n. apply* IOget.
Qed.

Haskell

`rpc w w' f n =` iOget w w' (f n)

Extraction

Lemma (Remote Procedure Call)

$\forall w\ w', (\mathbb{N} \to \mathtt{IO}\ w'\ \mathtt{bool}) \to (\mathbb{N} \to \mathtt{IO}\ w\ \mathtt{bool})$

Proof. simpl; introv f. intro n. appl* IO

Given two worlds w w'

Qed.

Haskell

```
rpc w w'  f  n  =   iOget w w' (f n)
```

Extraction

Lemma (Remote Procedure Call)

$\forall w\ w', (\mathbb{N} \to \mathtt{IO}\ w'\ \mathtt{bool}) \to (\mathbb{N} \to \mathtt{IO}\ w\ \mathtt{bool})$

Proof. simpl; introv f. intro n. apply* IOget.
Qed.

Given a function f

Haskell

```
rpc w w'  f  n  =   iOget w w' (f n)
```

Extraction

Lemma (Remote Procedure Call)

$\forall w\ w', (\mathbb{N} \to \mathit{IO}\ w'\ \mathit{bool}) \to (\mathbb{N} \to \mathit{IO}\ w\ \mathit{bool})$

Proof. simpl; introv f. intro n. apply* IOget.
Qed.

Given a value, say *n*

Haskell

```
rpc w w'  f  n  =   iOget w w' (f n)
```

Extraction

Lemma (Remote Procedure Call)

$\forall w\ w', (\mathbb{N} \to IO\ w'\ bool) \to (\mathbb{N} \to IO\ w\ bool)$

Proof. simpl; introv f. intro n. apply* IOget.
Qed.

Apply `IOget` to $(f\ n)$

(World parameters are inferred)

Haskell

```
rpc w w'  f  n  =   iOget w w' (f n)
```

Extraction

Lemma (Remote Procedure Call)

$\forall w\ w', (\mathbb{N} \to \mathit{IO}\ w'\ \mathit{bool}) \to (\mathbb{N} \to \mathit{IO}\ w\ \mathit{bool})$

Proof. simpl; introv f. intro n. apply* IOget.
Qed.

Haskell

```
rpc w w' f n =  iOget w w' (f n)
```

Notice: The annotated Haskell code is not runnable yet!

The HEC Compiler: the mobility fragment

$$\begin{aligned}\mathcal{M}[\![\texttt{iOget}\ A_1\ A_2\ (F\ A_3)]\!]_\rho = \ & \textbf{spawn}\texttt{(element(2, } \mathcal{E}[\![A_1]\!]_\rho\texttt{), } \rho(\eta)\texttt{,}\\ & \quad \texttt{dispatcher,[fun () -> } \mathcal{E}[\![F]\!]_\rho \mathcal{E}[\![A_3]\!]_\rho \textbf{end},\\ & \quad \mathcal{E}[\![A_2]\!]_\rho, \{\textbf{self}\texttt{(), } \textbf{node}\texttt{()}\}\texttt{]),}\\ & \textbf{receive}\{\texttt{result, Z}\}\texttt{ -> Z}\end{aligned}$$

Erlang Primitives remind:

`Pid` = **spawn** `(Host, Module, Function, Parameters)` (Code Mobilty)

receive {`Pattern`} `->` `Expression` (Receive)

`Pid !` Expression (Send)

The HEC Compiler: the location fragment

New

$$\mathcal{M}[\![\texttt{iOnew}\ A_1\ A_2\ A_3]\!]_\rho = \\ (\mathcal{E}[\![A_3]\!]_\rho)(\textbf{spawn}\texttt{(}\textbf{element}\texttt{(2, }\mathcal{E}[\![A_1]\!]_\rho\texttt{)},\rho\texttt{("module name"),} \\ \texttt{location, [}\mathcal{E}[\![A_2]\!]_\rho\texttt{]))}$$

Update

$$\mathcal{M}[\![\texttt{iOupdate}\ w\ A_1\ A_2\ A_3]\!]_\rho = \mathcal{E}[\![A_1]\!]_\rho!\{\textbf{update},\mathcal{E}[\![A_2]\!]_\rho\},\mathcal{E}[\![A_3]\!]_\rho$$

Lookup

$$\mathcal{M}[\![\texttt{iOlookup}\ w\ A_1\ A_2]\!]_\rho = \mathcal{E}[\![A_1]\!]_\rho\texttt{!}\{\mathrm{get},\{\textbf{self}(),\textbf{node}()\}\}, \\ \textbf{receive}\{\texttt{result, Z}\} \rightarrow (\mathcal{E}[\![A_2]\!]_\rho)(Z)$$

Erlang Primitives remind:

`Pid` = **spawn** `(Host, Module, Function, Parameters)` (Code Mobilty)
receive {`Pattern`} `-> Expression` (Receive)
`Pid` ! Expression (Send)

The `rpc` example

Haskell

```
rpc w w' f n = iOget w w' (f n)
```

Erlang

```
spawn(element(2, w), ρ(η),
      dispatcher,
[fun () -> f(n) end,w',{self(), node()}]),
receive{result, Z} -> Z
```

The `rpc` example

Haskell

```
rpc w w’ f n = iOget w w’ (f n)
```

Erlang

```
spawn(element(2, w), ρ(η),
      dispatcher,
[fun () -> f(n) end, w’, {self() , node()}]),
receive{result, Z} -> Z
```

$\mathcal{M}[\![\_]\!] : \mathbb{H} \to \mathbb{E}$

Haskell annotations are exploited by the HEC compiler

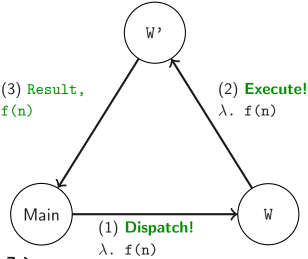
The `rpc` example

Haskell

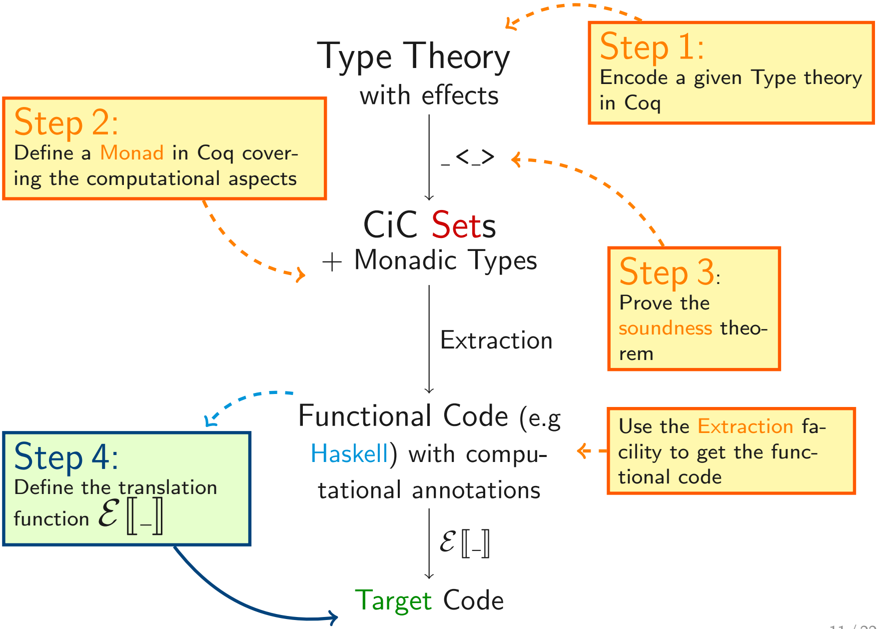
```
rpc w w' f n = iOget w w' (f n)
```

Erlang

```
spawn(element(2, w), ρ(η),
      dispatcher,
[fun () -> f(n) end,w',{self(), node()}]),
receive{result, Z} -> Z
```



Adding a Type Theory



Type Theory
with effects

_<_>

CiC Sets
+ Monadic Types

Extraction

Functional Code (e.g Haskell) with computational annotations

$\mathcal{E}[\![\_]\!]$

Target Code

Step 1:
Encode a given Type theory in Coq

Step 2:
Define a Monad in Coq covering the computational aspects

Step 3:
Prove the soundness theorem

Use the Extraction facility to get the functional code

Step 4:
Define the translation function $\mathcal{E}[\![\_]\!]$

λ_{XD} : A Type Theory for distributed computations

We give a Type theory similar to the Intuitionistic Hybrid Modal Logic of Licata and Harper[1]

Syntax (Hybrid Modal Logic IS5)

$$
\begin{array}{lll}
\text{Types} & A ::= & \text{Unit} \mid \text{Bool} \mid \text{Nat} \mid A \times B \mid A \rightarrow B \\
 & & \mid \text{Ref} \qquad\qquad (\text{Locations}) \\
 & & \mid \forall w.A \qquad\qquad (\text{Necessarily} \quad \Box A) \\
 & & \mid \exists w.A \qquad\qquad (\text{Possibly} \quad \Diamond A) \\
 & & \mid A @ w \qquad\qquad (\text{Nominal}) \\
 & & \mid \bigcirc A \qquad\qquad (\text{Lax Modality})
\end{array}
$$

---

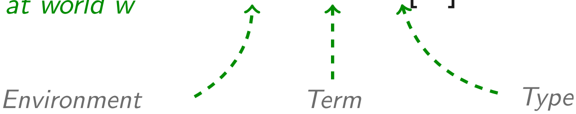
[1] "A monadic formalization of ML5". In Proc. LFMTP, EPTCS 34, 2010.

λ_{XD}: Type System

“*A* is inhabited with *M at world w*”

$$\Gamma \vdash M : A[w]$$

(Judgment)



Environment

Term

Type

▸ World Fragment

▸ Monadic Fragment

λ_{XD}: Type System

"A is inhabited with M at world w"

$$\Gamma \vdash M : A[w]$$

(Judgment)

Environment *Term* *Type*

World Fragment

(A small subset)

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \ \text{some } w \ M : \exists w.A[w]} \ (\text{some})$$

Intuitionistic $\exists$

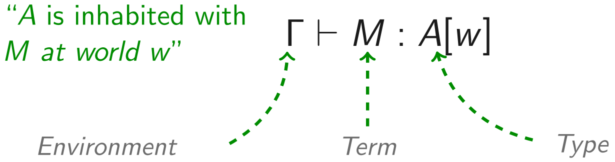
- a world w as Witness
- a Proof of $A(w)$

$$\frac{\Gamma \vdash M : \exists u.A[w] \quad \Gamma, x : A\{z/u\}[w] \vdash N : C[w'] \quad z \text{ fresh in } \Gamma}{\Gamma \vdash \ \text{letd } (z, x) = M \ \text{ in } N : C[w']} \ (\text{letd})$$

‣ World Fragment

‣ Monadic Fragment

λ_{XD}: Type System



“*A* is inhabited with *M* at world *w*”

$$\Gamma \vdash M : A[w]$$

(Judgment)

Environment

Term

Type

Monadic Fragment

(A small subset)

$$\frac{\text{Mobile } A \quad \Gamma \vdash M : \bigcirc A[w']}{\Gamma \vdash \text{ get } w' \; M : \bigcirc A[w]} \;(\text{Get})$$

World Fragment

Monadic Fragment

Soundness

We restrict the mobile types which are movable:

$$\frac{b \in \{\text{Unit, Nat, Bool}\}}{\text{Mobile } b}\ (\text{Basic}) \qquad \frac{\text{Mobile } A \quad \text{Mobile } B}{\text{Mobile } A \times B}\ (\text{Prod})$$

$$\frac{}{\text{Mobile } A@w}\ (\text{At}) \qquad \frac{\text{Mobile } A}{\text{Mobile } \forall w.A}\ (\text{Forall}) \qquad \frac{\text{Mobile } A}{\text{Mobile } \exists w.A}\ (\text{Exists})$$

Lemma (Mobility)

$\forall A \in$ Type, if Mobile A, then $\forall w,\ w' \in \mathtt{W}$, A <w> $=$ A <w'>.

`[Coq proof]`

λ_{XD}: Translation into CIC `Sets`

$$\_\,\texttt{<}w\texttt{>} : \text{Type}_{\lambda_{XD}} \to \texttt{Type}$$

$\text{Unit}\,\texttt{<w>} = \texttt{unit}$

$\text{Nat}\,\texttt{<w>} = \texttt{nat}$

$A \times B\,\texttt{<w>} = A\,\texttt{<w>}\ \texttt{*}\ B\,\texttt{<w>}$

$\text{Bool}\,\texttt{<w>} = \texttt{bool}$

$\text{Ref}\,\texttt{<w>} = \texttt{ref}\ w$

$A@w'\,\texttt{<w>} = A\,\texttt{<w'>}$

$A \to B\,\texttt{<w>} = A\,\texttt{<w>}\ \texttt{->}\ B\,\texttt{<w>}$

$\forall w'.A\,\texttt{<w>} = \texttt{forall w'}, (A\ w')\,\texttt{<w>}$

$\exists w'.A\,\texttt{<w>} = \{\ \texttt{w' : world}\ \&\ (A\ w')\,\texttt{<w>}\ \}$

$\bigcirc\, A\,\texttt{<w>} = \texttt{IO w}\ (A\,\texttt{<w>})$

Theorem (Soundness)

Let $\Gamma \in \mathit{Ctxt}$, $t \in \mathit{Term}$, $A \in \mathit{Type}$, $w \in \mathit{World}$, let $\{w_1, \ldots, w_n\}$ be all free worlds in Γ, A, t, w.

$$\Gamma \vdash_{XD} t : A[w] \quad \to \quad w_1 : W, \ldots, w_n : W \vdash \_ : [\![\Gamma]\!] \to A\,\texttt{<}w\texttt{>}.$$

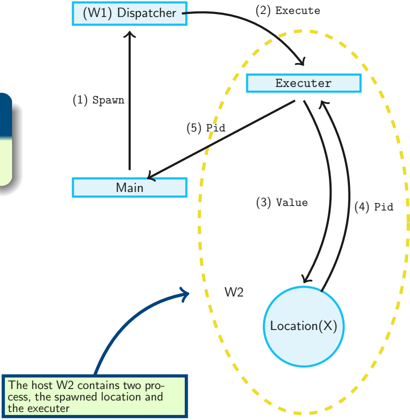
`[Coq proof]`

Example: Remote Write

Lemma (Remote Write)

$\forall w' \;\; w'', \mathbb{N}@w' \rightarrow_{XD} \bigcirc(\mathit{Ref}@w'')\ \text{<}w'\text{>}$

Proof. simpl; introv value. apply IOget with (remote := w2). apply IOnew. exact value. intros address. apply IOret. exact address. **Defined**.



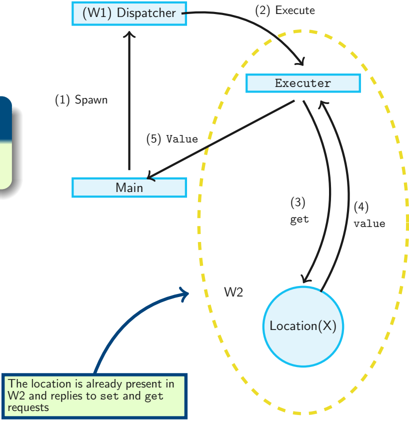
The host W2 contains two process, the spawned location and the executer

Example: Remote Read

Lemma (Remote Read)

$\forall w' \ w'', Ref @ w'' \rightarrow_{XD} \bigcirc(\mathbb{N}) <w'>$

Proof. simpl; introv address. apply IOget with (remote := w2). apply* IOlookup. intros. apply* IOret.
Defined.



The location is already present in W2 and replies to set and get requests

Update Lemma

A suitable lemma can be stated and proved in order to guarantee the system behaves correctly

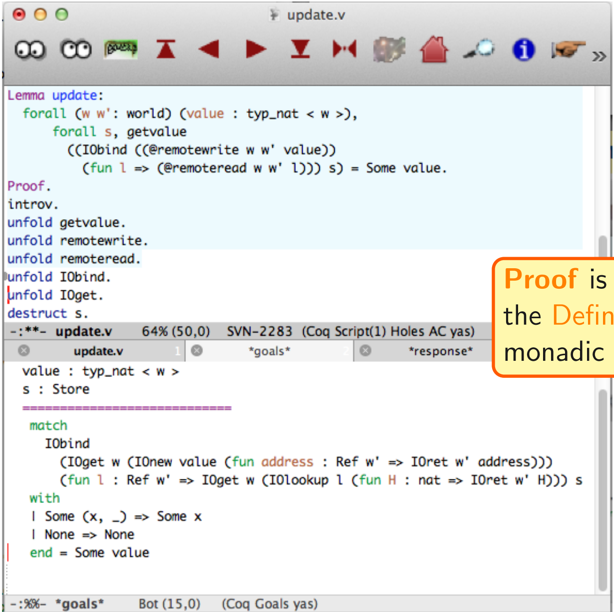
Lemma (Update operation is correct)

$\forall$*w w'*,$\forall\mathbb{N}$, $\forall s$: *Store*,

```
getvalue ((IObind ((@remotewrite w w' value ))
(λ l.  (@remoteread w w' l))) s) = Some value.
```

[Coq proof]

Update lemma: proof deals with monadic constructors



```
Lemma update:
  forall (w w': world) (value : typ_nat < w >),
      forall s, getvalue
        ((IObind ((@remotewrite w w' value))
          (fun l => (@remoteread w w' l))) s) = Some value.
Proof.
introv.
unfold getvalue.
unfold remotewrite.
unfold remoteread.
unfold IObind.
unfold IOget.
destruct s.
-:**-  update.v      64% (50,0)    SVN-2283  (Coq Script(1) Holes AC yas)
```

```
  value : typ_nat < w >
  s : Store
  ============================
   match
     IObind
       (IOget w (IOnew value (fun address : Ref w' => IOret w' address)))
       (fun l : Ref w' => IOget w (IOlookup l (fun H : nat => IOret w' H))) s
   with
   | Some (x, _) => Some x
   | None => None
   end = Some value

-:%%-  *goals*      Bot (15,0)    (Coq Goals yas)
```

Proof is by unfolding the Definitions of the monadic operators

Conclusions

Summary

- We presented a generic methodology for code extraction which works in principle with every computational aspect
  - + Define a front-end type theory on top (optional)
  - + Define a `IO` monad over `Set`
  - + Implement the back-end compiler
- We shown how to extract distributed code by defining a Distributed Monad `IO w A`
- We defined the λ_{XD} on top

Conclusions

Future work

- Other computational aspects / target languages
- is the compiler correct? ⇒ Compiler Certification could be achieved by using the implementation of monadic operators as the specification of how the target code must behave
- Alternatively, an Axiomatization for the operators is needed
  - + which takes into account the mobility
  - + not available yet (AFAIK)
  - + but could be achived extending similar work (see e.g., Power and Plotkin ▸ Axioms)

Thanks for your attention.
Questions?

Appendix: λ_{XD} Typesystem - World Fragment

$$\frac{\Gamma \vdash M : \ A[w'] \quad w \text{ fresh in } \Gamma}{\Gamma \vdash \Lambda w.M : \forall w.A[w']} \text{(Box)} \qquad \frac{\Gamma \vdash M : \forall w.A[w]}{\Gamma \vdash \ \text{unbox } w' M : A[w]} \text{(Unbox)}$$

$$\frac{\vdash \Gamma \quad x : A[w] \in \Gamma}{\Gamma \vdash x : A[w]} \text{(Var)} \qquad \frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \ \text{some } w \ M : \exists w.A[w]} \text{(Some)}$$

$$\frac{\Gamma \vdash M : \exists u.A[w] \quad \Gamma, x : A\{z/u\}[w] \vdash N : C[w'] \quad z \text{ fresh in } \Gamma}{\Gamma \vdash \ \text{letd } (z, x) = M \ \text{ in } N : C[w']} \text{(LetD)}$$

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \text{hold } M : A@w[w']} \text{(Hold)}$$

$$\frac{\Gamma \vdash M : A@z[w] \quad \Gamma, x : A[z] \vdash N : C[w]}{\Gamma \vdash \ \text{leta } x = M \text{ in } N : C[w]} \text{(LetA)}$$

▸ Return

Appendix: λ_{XD} Typesystem - Monadic fragment

$$\frac{\Gamma \vdash M : A[w]}{\Gamma \vdash \text{ return } M : \bigcirc A[w]} \text{(Ret)} \qquad \frac{\Gamma \vdash M : \bigcirc A[w] \quad \Gamma \vdash N : A \to \bigcirc B[w]}{\Gamma \vdash \text{ bind } M\ N : \bigcirc B[w]}$$

$$\frac{\text{Mobile } A \quad \Gamma \vdash M : \bigcirc A[w']}{\Gamma \vdash \text{ get } w'\ M : \bigcirc A[w]} \text{(Get)}$$

$$\frac{\Gamma \vdash M : \text{Nat}[w] \quad \Gamma \vdash N : \text{Ref} \to \bigcirc A[w]}{\Gamma \vdash \text{ new } M\ N : \bigcirc A[w]} \text{(New)}$$

$$\frac{\Gamma \vdash M : \text{ Ref } [w] \quad \Gamma \vdash N : \text{Nat} \to \bigcirc A}{\Gamma \vdash \text{ lookup } M\ N : \bigcirc A[w]} \text{(Lookup)}$$

$$\frac{\Gamma \vdash T1 : \text{ Ref } [w] \quad \Gamma \vdash T2 : \text{Nat}[w] \quad \Gamma \vdash T3 : \bigcirc A[w]}{\Gamma \vdash \text{ update } T1\ T2\ T3 : \bigcirc A[w]} \text{(Update)}$$

▸ Return

Appendix: On the axiomatization

Rules for the `lookup` and `update` are borrowed from Plotkin and Power [1]:

1.$l_{loc}(u_{loc,v}(x))v = x$

2.$l_{loc}(l_{loc}(t_{vv'})_v)_{v'} = l_{loc}(t_{vv})_v$

3.$u_{loc,v}(u_{loc,v'}(x)) = u_{loc,v'}(x)$

4.$u_{loc,v}(l_{loc}(t_{v'})_{v'}) = u_{loc,v}(t_v)$

5.$l_{loc}(l'_{loc}(t_{vv'})_{v'})_v = l'_{loc}(l_{loc}(t_{vv'})_v)_{v'}$where $loc = loc'$

6.$u_{loc,v}(u_{loc',v'}(x)) = u_{loc',v'}(u_{loc,v}(x))$where $loc = loc''$

▸ Return

---

[1] G. D. Plotkin and J. Power. Notions of computation determine monads. In Proc. FoSSaCS, LNCS 2303, 2002.