MANAGING LOGICAL AND COMPUTATIONAL COMPLEXITY USING PROGRAM TRANSFORMATIONS
From complex model to simple model

Build complex models from simpler ones.

- Presheaves
- Grothendieck Sheaves
- Coalgebras
- Quotients
From complex model to simple model

Example: Presheaves

$F : \text{Presheaves}$

$F : A \rightarrow \text{Set}$
From high-level languages to low-level languages

Compile high level languages into low level languages

- C → binary code
- Prolog → abstract machine code
- Scala → Java bytecode
- Defunctionalization
From high-level languages to low-level languages

Example:

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```
Limitations of the model approach

The complex theory needs to be implemented

- too much work

or

axioms must be stated in the simple theory

- axioms have no computational meaning
From complex logics to simple logics

Gödel–Gentzen **translation** from classical logic to intuitionistic logic
From complex logics to simple logics

Gödel-Gentzen *translation* from classical logic to intuitionistic logic

Through the Curry-Howard correspondance:

A compilation phase from classical programs to intuitionistic programs
From complex logics to simple logics

Gödel-Gentzen translation from classical logic to intuitionistic logic

$\phi^N \rightarrow \neg\neg\phi$

$(\phi \land \theta)^N \rightarrow \phi^N \land \theta^N$

$(\phi \lor \theta)^N \rightarrow \neg(\neg\phi^N \land \neg\theta^N)$

$LK = LJ + \text{Ex Mid}$

$LJ$
From complex logics to simple logics

Through the Curry-Howard correspondance:

Logical translation = program compilation
Claim

The connection between extension of models and compilation of languages is more than an analogy.
The connection between extension of models and compilation of languages is more than an analogy.

The target system is the type theory of Coq, seen as an assembly language of logic.
Methodology. Distinct compilation phases

Compile complex type theories into simpler ones.

- inherit consistency of Coq
- split the complexity of type checking
Extending Type Theory

with effects
The relativity of truth

“Truth (respectively, membership in a type) is not absolute but relative to some appropriate notion of world, or state, and that these states can be related precisely to each other.”

(A Very Modal Model of a Modern, Major, General Type System, POPL’07)

Forcing translation \(\cong\) Kripke models \(\cong\) Presheaf construction

(Set theory)  (Modal logic)  (Category theory)
The relativity of truth

Forcing translation \(\cong\) Kripke models \(\cong\) Presheaf construction

\((Set\ theory)\) \((Modal\ logic)\) \((Category\ theory)\)

A formula \(\varphi\) is true when \(\omega \models \varphi\) is true for every possible “world” \(\omega\).
The relativity of values

In functional programming, monads are a way to express programming concept such as states, exceptions, non-determinism.
The relativity of values

In functional programming, monads are a way to express programming concepts such as states, exceptions, non-determinism.

The state monad.

The interpretation of an arrow is enriched as:

\[ [A \rightarrow B] \equiv S \times A \rightarrow S \times B \]

Monadic translation/compilation allows to give a meaning to memory in functional programming.
The relativity of values

The state monad.

The interpretation of an arrow is enriched as:

\[ [A \to B] \equiv S \times A \to S \times B \]

Now, the return value of a function may depend on the current state.
The relativity of values

The state monad.

The interpretation of an arrow is enriched as:

\[[A \rightarrow B] \equiv S \times A \rightarrow S \times B\]

Now, the return value of a function may depend on the current state.

<table>
<thead>
<tr>
<th>Forcing / Presheaves</th>
<th>State monad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relativity of truth</td>
<td>Relativity of values</td>
</tr>
</tbody>
</table>
Idea

Use the monadic compilation to implement forcing in type theory

source theory

Coq + general fixpoints

forcing

target theory

Coq
Idea

Use the monadic compilation to implement forcing in type theory

**Problem:** The monadic translation does not lift to type theory because of dependencies!
On the order of evaluation

There are two main kinds of evaluation strategy:

- Call-by-name. Functions are evaluated first
- Call-by-value. Arguments are evaluated first
On the order of evaluation

There are two main kinds of evaluation strategy:

- Call-by-name. Functions are evaluated first
- Call-by-value. Arguments are evaluated first

The monadic translation is call-by-value, type theory is call-by-name!
Wait, what does Pim mean here?
Call-By-Push-Value

CBV

CBN

CBPV
Call-By-Push-Value

CBPV distinguishes between value and computation directly in the syntax.

value types

computation types

value terms

computation terms

\[ A, B ::= \forall X \mid \alpha \]

\[ X, Y ::= A \to X \mid \mathcal{F} A \]

\[ v ::= x \mid \text{thunk } t \]

\[ t, u ::= \lambda x : A. t \mid \mathsf{let } x : A := t \mathsf{ in } u \mid \text{force } t \mid \text{return } v \]
Call-By-Push-Value

CBPV distinguishes between value and computation directly in the syntax.

\[
\begin{align*}
A, B & ::= \mathcal{U} X \mid \alpha \\
X, Y & ::= \_ A \to X \mid \mathcal{F} A \\
v & ::= x \mid \text{thunk } t \\
t, u & ::= \lambda x : A. t \mid t v \mid \\
& \quad \text{let } x : A ::= t \text{ in } u \mid \\
& \quad \text{force } t \mid \text{return } v
\end{align*}
\]
From CBV to CBPV

\[ [A \to B]^v := U([A]^v \to F[B]^v) \]

We recover the monadic translation
From CBV to CBPV

\[ [A \rightarrow B]^v := U([A]^v \rightarrow F[B]^v) \]

We recover the monadic translation:

For the state monad, take:

\[ UA := S \rightarrow A \]
\[ FA := S \times A \]
From CBV to CBPV

\[ [A \rightarrow B]^v := \mathcal{U}([A]^v \rightarrow \mathcal{F}[B]^v) \]

We recover the monadic translation

For the state monad, take:

\[ \mathcal{U}A := S \rightarrow A \]
\[ \mathcal{F}A := S \times A \]

because

\[ S \rightarrow (A \rightarrow S \times B) \]
\[ \sim \]
\[ S \times A \rightarrow S \times B \]
From CBV to CBPV

CBV → CBN → CBPV → λΠ
From CBV to CBPV

does not preserve conversion
From CBN to CBPV

\[
[A \rightarrow B]^n := \mathcal{U} [A]^n \rightarrow [B]^n
\]
From CBN to CBPV

\[ [A \rightarrow B]^n := U [A]^n \rightarrow [B]^n \]
From CBN to CBPV

\[ [A \to B]^n := \mathcal{U} [A]^n \to [B]^n \]

does preserve conversion
Decomposing the monad as an adjunction: version 1

\[ \text{Obj}(\mathcal{C}_T) = \text{Obj}(\mathcal{C}), \]
\[ \text{Hom}_{\mathcal{C}_T}(X, Y) = \text{Hom}_{\mathcal{C}}(X, TY) \]
Decomposing the monad as an adjunction : version I

\[ u_T X = TX \quad F_T X = X \]
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
\begin{align*}
[U \, X]_p & := \prod q : \mathbb{P}. \, p \leq q \to [X]_q \\
[\mathcal{F} \, A]_p & := [A]_p
\end{align*}
\]

CBV decomposition.

\[
\mathcal{U}([A]^v \to \mathcal{F} \, [B]^v)
\]
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
[\mathcal{U} X]_p := \Pi q : \mathbb{P}. p \leq q \rightarrow [X]_q
\]

\[
[\mathcal{F} A]_p := [A]_p
\]

CBV decomposition.

\[
\Pi q : \mathbb{P}. p \leq q \rightarrow [A]_q \rightarrow [B]_q
\]
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
\begin{align*}
[U X]_p &:= \Pi q : \mathbb{P}. p \leq q \rightarrow [X]_q \\
[F A]_p &:= [A]_p
\end{align*}
\]

CBV decomposition.

\[
\Pi q : \mathbb{P}. p \leq q \rightarrow [A]_q \rightarrow [B]_q
\]

Provide the usual presheaf interpretation of forcing
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
[U X]_p := \Pi q : P. p \leq q \rightarrow [X]_q
\]

\[
[F A]_p := [A]_p
\]

CBV decomposition.

\[
\Pi q : P. p \leq q \rightarrow [A]_q \rightarrow [B]_q
\]

Provide the usual presheaf interpretation of forcing.
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
\begin{align*}
[\mathcal{U} X]_p & := \Pi q : \mathbb{P}. p \leq q \rightarrow [X]_q \\
[\mathcal{F} A]_p & := [A]_p
\end{align*}
\]

CBN decomposition.

\[
\mathcal{U} [A]^n \rightarrow [B]^n
\]
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
\begin{align*}
[U \ X]_p & \ := \ \Pi q : \mathbb{P}. \ p \leq q \ \rightarrow \ [X]_q \\
[F \ A]_p & \ := \ [A]_p
\end{align*}
\]

CBN decomposition.

\[
[\Pi x : A. \ B]_p \equiv \Pi (x : \Pi q \leq p. \ [A]_q). \ [B]_p
\]
Forcing in Type Theory

We take the Kleisli adjunction for the (local) reader monad

\[
\begin{align*}
[\mathcal{U} \, X]_p & := \Pi q : \mathbb{P}. p \leq q \rightarrow [X]_q \\
[\mathcal{F} \, A]_p & := [A]_p
\end{align*}
\]

CBN decomposition.

\[
[\Pi x : A. \, B]_p \equiv \Pi (x : \Pi q \leq p. [A]_q). [B]_p
\]
Forcing in Type Theory

\[ [\Box_i]_\sigma := \lambda(q\, f : \sigma). \Pi(r\, g : \sigma \cdot (q, f)) \cdot [\Box_i]_\sigma \]

\[ [x]_\sigma := x \sigma_e \sigma(x) \]

\[ [\lambda x : A. M]_\sigma := \lambda x : [A]_\sigma^! \cdot [M]_\sigma^! \cdot x \]

\[ [M \cdot N]_\sigma := [M]_\sigma \cdot [N]_\sigma^! \]

\[ [\Pi x : A. B]_\sigma := \lambda(q\, f : \sigma). \Pi x : [A]_\sigma^! \cdot (q, f) \cdot [B]_\sigma^! \cdot (q, f) \cdot x \]

\[ [A]_\sigma := [A]_\sigma \sigma_e \cdot \text{id}_{\sigma_e} \]
Forcing in Type Theory

\[ [\square_i]_\sigma := \lambda(qf : \sigma) \cdot \Pi(rg : \sigma \cdot (q, f)) \cdot \square_i \]
\[ [x]_\sigma := x \sigma_e \sigma(x) \]
\[ [\lambda x : A. M]_\sigma := \lambda x : [A]_\sigma^! \cdot [M]_\sigma \cdot x \]
\[ [M N]_\sigma := [M]_\sigma \cdot [N]^!_\sigma \]
\[ [\Pi x : A. B]_\sigma := \lambda(qf : \sigma) \cdot \Pi x : [A]_\sigma^! \cdot (q, f) \cdot [B]_\sigma \cdot (q, f) \cdot x \]
\[ [A]_\sigma := [A]_\sigma \sigma_e \cdot \text{id}_{\sigma_e} \]

**Theorem 1** (Computational Soundness). If \( \Gamma \vdash M \equiv N \) then \( [\Gamma]_p \vdash [M]_p \equiv [N]_p \).
Forcing in Type Theory

\[
\begin{align*}
[\square_i]_\sigma & \quad ::= \quad \lambda(q \, f : \sigma). \Pi(r \, g : \sigma \cdot (q, f)). \square_i \\
[x]_\sigma & \quad ::= \quad x \, \sigma_e \, \sigma(x) \\
[\lambda x : A. \, M] & \quad ::= \quad \lambda x : [A]! . [M] \\
[M \, N]_\sigma & \quad ::= \quad [A]_\sigma \, \sigma_e \, \text{id}_{\sigma_e} \\
[\Pi x : A. \, M] & \quad ::= \quad \Pi x : [A]! . [M]
\end{align*}
\]

Allow for a direct implementation in Coq!

**Theorem 1** (Computational Soundness). If \( \Gamma \vdash M \equiv N \) then \( [\Gamma]_p \vdash [M]_p \equiv [N]_p \).
The definitional side of the Forcing

Abstract
This paper studies forcing translations of proofs in dependent type theory, through the Curry-Howard correspondence. Based on a call-by-push-value decomposition, we synthesize two simply-typed translations: i) one call-by-value, corresponding to the translation derived from the presheaf construction as studied in a previous paper; ii) one call-by-name, whose intuitions already appear in Krivine and Miquel’s work. Focusing on the call-by-name translation, we adapt it to the dependent case and prove that it is compatible with the definitional equality of our system, thus avoiding coherence problems. This allows us to use any category as forcing conditions, which is out of reach with the call-by-value translation. Our construction also exploits the notion of storage operators in order to interpret dependent elimination for inductive types. This is a novel example of a dependent theory with side-effects, clarifying how dependent elimination for inductive types must be restricted in a non-pure setting. Being implemented as a Coq plugin, this work gives the possibility to formalize easily consistency results, for instance the consistency of the negation of Voevodsky’s univalence axiom.

Categories and Subject Descriptors
F.4.1 [MATHEMATICAL LOGIC AND FORMAL LANGUAGES]: Mathematical Logic
Keywords
Forcing, Dependent type theory, Inductive types, EF-theoretic metaprogramming.

It is then straightforward to extend the construction to work on categories of forcing conditions, rather than simply posets, giving a proof-relevant version of forcing.

Recent years have seen a renewal of interest for forcing, driven by Krivine’s classical realizability [9]. In this line of work, forcing is studied as a proof translation, and one seeks to understand its computational content [3, 12], through the Curry-Howard correspondence. This means that 

\[ \forall \beta \exists \alpha \] is studied as a syntactic translation of formulas, parametrized by a forcing condition \( p \). Following these ideas, a forcing translation has been defined in [6] for the Calculus of Constructions, the type theory behind the Coq proof assistant. It is based heavily on the presheaf construction of Lawvere and Tierney. The main goal of [6] was to extend the logic behind Coq with new principles, while keeping its fundamental properties: soundness, canonicity and decidability of type checking. This approach can be seen, following [1], as type-theoretic metaprogramming.

However, this technique suffers from coherence problems, which complicate greatly the translation. More precisely, the translation of two definitionally equal terms are not in general definitionally equal, but only propositionally equal. Rewriting terms must then be inserted inside the definition of the translation. If this is possible to perform, albeit tedious, when the forcing conditions form a poset, it becomes intractable when we want to define a forcing translation parametrized by a category of forcing conditions.

In this paper, we propose a novel forcing translation for the Calculus of Inductive Constructions (CIC), which avoids these coherence problems. Departing from the categorical intuitions of the presheaf construction, it takes its roots as a call-by-push-value [10] decomposition of our system. This will justify to name our translation call-by-name, while the previous translation of [6] is call-by-value.

"Call-by-name forcing provides the first effectful translation of \( \lambda \)-terms into \( \lambda \)-terms which preserves definitional equality."
Enlarge your Coq with Forcing

with new principles, without axioms
Enlarge your Coq with Forcing

= +

with new principles, without axioms

https://github.com/CoqHott/coq-forcing
Example: negation of univalence

Definition Obj := $\mathbb{B}$.
Definition Hom (p q : Obj) := unit.

Definition $A_0 := \lambda \ p \ q \ (a : p \leq q) \ r \ (b : q \leq r) \Rightarrow \text{if } r \text{ then } \top \text{ else } \bot$.
Definition $A_1 := \lambda \ p \ q \ (a : p \leq q) \ r \ (b : q \leq r) \Rightarrow \text{if } r \text{ then } \bot \text{ else } \top$.

Forcing Definition neg_univ : univalence $\Rightarrow \bot$ using Obj Hom.
Proof.
intros p huniv.

(* Definition of the equivalence function and its inverse *)

refine (let f := _ : \forall (p0 : Obj) (a : p \leq p0),
\forall (p1 : Obj) (a0 : p0 \leq p1),
A0 p p1 (a \circ a0) p1 #) \rightarrow A1 p p0 a p0 # in _).
intros. specialize (H (neg p0) (\_ \_ ⇒ tt)). destruct p0; exact H.
refine (let g := _ : \forall (p0 : Obj) (a : p \leq p0),
\forall (p1 : Obj) (a0 : p0 \leq p1),
A1 p p1 (a \circ a0) p1 #) \rightarrow A0 p p0 a p0 # in _).
intros. specialize (H (neg p0) (\_ \_ ⇒ tt)). destruct p0; exact H.
Example: negation of univalence

```
Definition Obj := ℕ.
Definition Hom (p q : Obj) := unit.

Definition A₀ := λ p q (a : p ≤ q) r (β : q ≤ r) ⇒ if r then TR else T.
Definition A₁ := λ p q (a : p ≤ q) r (β : q ≤ r) ⇒ if r then T else TR.

Forcing Definition neg_univ : univalence → ⊥ using Obj Hom.
```

```
Proof.
intros p huniv.

(* Definition of the equivalence function and its inverse *)

refine (let f := _ : ∀ (p₀ : Obj) (a : p ≤ p₀),
  (∀ (p₁ : Obj) (a₀ : p₀ ≤ p₁),
   A₀ p p₁ (a ⊖ a₀) p₁ #) → A₁ p p₀ a p₀ # in _).
intros. specialize (H (neg p₀) (λ _ _ ⇒ tt)). destruct p₀; exact H.
refine (let g := _ : ∀ (p₀ : Obj) (a : p ≤ p₀),
  (∀ (p₁ : Obj) (a₀ : p₀ ≤ p₁),
   A₁ p p₁ (a ⊖ a₀) p₁ #) → A₀ p p₀ a p₀ # in _).
intros. specialize (H (neg p₀) (λ _ _ ⇒ tt)). destruct p₀; exact H.
```

Example: negation of univalence

Demo
What is missing to make it really usable

A definitionally proof irrelevant version of Coq with Prop ⊆ HProp.

A mechanism to lift definitional equality in the target to definitional equality in the source.
Example: realizing univalence?

Can we use the cube or simplex category to realize Cubical TT in Coq?

Maybe, but certainly with extra definitional equalities.
Effects and dependency

In this effectful setting, the usual dependent elimination of CIC is not valid!

Intuitively, this is because the general dependent elimination requires the evaluation of values to be delayed without changing the computation.
Effects and dependency

In this effectful setting, the usual dependent elimination of CIC is not valid!

Intuitively, this is because the general dependent elimination requires the evaluation of values to be delayed without changing the computation

Not true in presence of effects!
Krivine’s Storage Operators

Storage operators are a trick to encode call-by-value in a call-by-name setting thanks to a CPS.
Krivine’s Storage Operators

Storage operators are a trick to encode call-by-value in a call-by-name setting thanks to a CPS.

It allows to recover a restricted form of dependent elimination.
Storage Operator for $\Sigma$

$\Gamma \vdash M : \Sigma x : A. B \quad \Gamma, z : \Sigma x : A. B \vdash C : \square \quad \Gamma, x : A, y : B \vdash N : C\{z := (x, y)\}$

$\Gamma \vdash \text{match } M \text{ with } (x, y) \Rightarrow N : \theta_{\Sigma} M \, (\lambda z : \Sigma x : A. B. C')$
Storage Operator for $\Sigma$

\[
\begin{align*}
\Gamma \vdash M : \Sigma x : A. B & \quad \Gamma, z : \Sigma x : A. B \vdash C : \Box & \quad \Gamma, x : A, y : B \vdash N : C \{z := (x, y)\} \\
\hline
\Gamma \vdash \text{match } M \text{ with } (x, y) \Rightarrow N : \text{match } M \text{ with } (x, y) \Rightarrow C \{z := (x, y)\}
\end{align*}
\]
Storage Operator for lists

\[ \text{list}_\text{case} : \Pi A. P. P \to (A \to \text{list } A \to P \to P) \to \text{list } A \to P \]

\[ \theta_{\text{list}} : \Pi A. \text{list } A \to (\text{list } A \to \Box) \to \Box \]
\[ := \lambda A l k. \text{list}_\text{case } A ((\text{list } A \to \Box) \to \Box) \\
(\lambda k. k \text{ nil}) \\
(\lambda x_r r k. r (\lambda l. k (\text{cons } A x l))) l k \]
Storage Operator for lists

\[
\text{list\_rect}: \Pi A (P : \text{list} \ A \rightarrow \Box). P \ (\text{nil} \ A) \rightarrow \\
(\Pi x l. \theta_{\text{list}} A \ l \ P \rightarrow \theta_{\text{list}} A \ (\text{cons} \ A \ x \ l) \ P) \rightarrow \\
\Pi l : \text{list} \ A. \theta_{\text{list}} A \ l \ P
\]
Baclofen Type Theory

CIC has one notion of pattern-matching that implements both non-dependent and dependent elimination
CoqHoTT, a brand-new proof assistant based on Homotopy Type Theory

Baclofen Type Theory

CIC has one notion of pattern-matching that implements both non-dependent and dependent elimination

BTT has two:

- The first one is non-dependent and is unconditionally valid
- The second one is dependent and restricted through the use of storage operators.
Linearity to the rescue

We can use a linearity condition to remove this restriction

See the talk of Paul Blain Levy on Thursday (base on the work of Guillaume Munch)
Claim/Thesis

BTT models effectful type theories
Claim/Thesis

BTT models effectful type theories

We will now see that it works also to interpret another kind of effectful translation.
Decomposing the monad as an adjunction: version II

\[
\text{Obj}(C^T) = T\text{-algebras} \\
\text{Hom}_{C^T}(X, Y) = \text{morphisms of } T\text{-algebras}
\]
Decomposing the monad as an adjunction: version II

\[ U^T X = X \quad \text{and} \quad F^T X = T X \]
Monadic Translation of Type Theory

We take the Eilenberg-Moore adjunction

CBV decomposition.

\[ U([A]^v \rightarrow F[B]^v) \]
Monadic Translation of Type Theory

We take the Eilenberg-Moore adjunction

CBV decomposition.

\[ [A]_v \rightarrow T [B]_v \]
Monadic Translation of Type Theory

We take the Eilenberg-Moore adjunction

CBV decomposition.

\[[A]_v \rightarrow T [B]_v\]
Monadic Translation of Type Theory

We take the Eilenberg-Moore adjunction

CBV decomposition.

\[[A]_v \rightarrow T [B]_v\]

Provide the usual monadic interpretation of forcing
Monadic Translation of Type Theory

We take the Eilenberg-Moore adjunction

CBV decomposition.

\[ [A]_\nu \rightarrow T [B]_\nu \]

Provide the usual monadic interpretation of forcing
Monadic Translation of Type Theory

We take the Kleisli adjunction for the (local) reader monad

CBN decomposition.

\[ \mathcal{U} [A]^n \rightarrow [B]^n \]
Monadic Translation of Type Theory

We take the Kleisli adjunction for the (local) reader monad

CBN decomposition.
Monadic Translation of Type Theory

We take the Kleisli adjunction for the (local) reader monad

CBN decomposition.

\[[A]_V \rightarrow [B]_V\]
Wait, this is the identity translation?
Wait, this is the identity translation?

No, because we need to work in the universe of algebras!
Self-Algebraic Monad

The universe of algebras can simply be stated by the following dependent type

\[ \Box_i = \Sigma A : \Box_i. T A \to A. \]

Rk: No need for coherence condition because the translation is computational
Self-Algebraic Monad

In Type Theory, we have:

\[ \Box_i : \Box_{i+1} \]
Self-Algebraic Monad

In Type Theory, we have:

\[ \square_i : \square_{i+1} \]

The universe of algebras must be itself algebraic

\[ \square_i : \square_{i+1} \]
Self-Algebraic Monad

A self-algebraic proto-monad is given by

- \( T : \Box_i \rightarrow \Box_i \)
- \( \text{ret} : \Pi A : \Box_i. A \rightarrow T A \)
- \( \text{bnd} : \Pi(A : \Box_i)(B : \Box_j). T A \rightarrow (A \rightarrow T B) \rightarrow T B \)
- \( \text{El} : T \Box_i \rightarrow \Box_i \)
- \( \text{hbdn} : \Pi(A : \Box_i)(B : T \Box_j). T A \rightarrow (A \rightarrow (\text{El } B)\cdot \pi_1) \rightarrow (\text{El } B)\cdot \pi_1 \)
Self-Algebraic Monad

A self-algebraic proto-monad is given by

usual return and bind constructors

\begin{itemize}
\item $T : \Box_i \rightarrow \Box_i$
\item $\text{ret} : \Pi A : \Box_i . A \rightarrow T A$
\item $\text{bnd} : \Pi (A : \Box_i) (B : \Box_j) . T A \rightarrow (A \rightarrow T B) \rightarrow T B$
\item $\text{El} : T \Box_i \rightarrow \Box_i$
\item $\text{hbind} : \Pi (A : \Box_i) (B : T \Box_j) . T A \rightarrow (A \rightarrow (\text{El} B).\pi_1) \rightarrow (\text{El} B).\pi_1$
\end{itemize}
Self-Algebraic Monad

A self-algebraic proto-monad is given by

self-algebraic part

- \( T : \Box_i \rightarrow \Box_i \)
- \( \text{ret} : \Pi A : \Box_i. A \rightarrow T A \)
- \( \text{bnd} : \Pi (A : \Box_i)(B : \Box_i). T A \rightarrow (A \rightarrow T B) \rightarrow T B \)
- \( \text{El} : T \Box_i \rightarrow \Box_i \)
- \( \text{monad} : \Pi (A : \Box_i)(B : \Box_i). T A \rightarrow (A \rightarrow (\text{El} B) \cdot \pi_1) \rightarrow (\text{El} B) \cdot \pi_1 \)
A self-algebraic proto-monad is given by

\[ \begin{align*}
\bullet & \quad T : \Box_i \rightarrow \Box_i \\
\bullet & \quad \text{ret} : \Pi A : \Box_i . A \rightarrow T A \\
\bullet & \quad \text{bnd} : \Pi (A : \Box_i) (B : \Box_j) . T A \rightarrow (A \rightarrow T B) \rightarrow T B \\
\bullet & \quad \text{El} : T \Box_i \rightarrow \Box_i \\
\bullet & \quad \text{hbnd} : \Pi (A : \Box_i) (B : T \Box_j) . \\
& \quad \quad T A \rightarrow (A \rightarrow (\text{El} B) . \pi_1) \rightarrow (\text{El} B) . \pi_1
\end{align*} \]
Self-Algebraic Monad

A self-algebraic proto-monad is given by

- \( T : \Box_i \rightarrow \Box_i \)
- \( \text{ret} : \Pi A : \Box_i. A \rightarrow T A \)
- \( \text{bnd} : \Pi (A : \Box_i)(B : \Box_j). T A \rightarrow (A \rightarrow T B) \rightarrow T B \)
- \( \text{El} : T \Box_i \rightarrow \Box_i \)
- \( \text{hbnd} : \Pi (A : \Box_i)(B : T \Box_j). \\
\hspace{1cm} T A \rightarrow (A \rightarrow (\text{El} B). \pi_1) \rightarrow (\text{El} B). \pi_1 \)
Self-Algebraic Monad

Remark: hbind could be defined by using bind but are translation requires few definitional laws like

\[ \text{El} (\text{ret} \uplus_i M) \equiv M \]

but we don’t require all usual laws of monads to hold definitionally (nor propositionally)
The translation

\[
\begin{align*}
[\square_i] &:= \text{ret } [\square_{i+1} ((T \square_i), \mu_{\square_i})] \\
[x] &:= x \\
[\lambda x : A. M] &:= \lambda x : [A]. [M] \\
[M \ N] &:= [M] [N] \\
[\Pi x : A. B] &:= \text{ret } \square ((\Pi x : [A]. [B]), \mu_{\Pi A B}) \\
[A] &:= (\text{El } [A]).\pi_1
\end{align*}
\]
The translation

\[\begin{align*}
\mu \Box_i & \quad := \quad \lambda A : T \ (T \ \Box_i) . \\
& \phantom{:=} \hbox{bnd} \ (T \ \Box_i) \ \Box_i A \ (\lambda X : T \ \Box_i . X)
\end{align*}\]

\[\begin{align*}
\mu \Pi A B & \quad := \quad \lambda (\hat{f} : T \ (\Pi x : [A]. [B])) \ (x : [A]) . \\
& \phantom{:=} \hbox{hbd} \ (\Pi x : [A]. [B]) \ [B] \\
& \phantom{:=} \hat{f} \ (\lambda f : \Pi x : [A]. [B]. f \ x)
\end{align*}\]
The translation

\[ \mu_{\box_i} := \lambda A : T \ (T \ \box_i) . \]
\[ \text{bnd} \ (T \ \box_i) \ \box_i \ A \ (\lambda X : T \ \box_i . \ X) \]

\[ \mu_{\Pi_{AB}} := \lambda (\hat{f} : T \ (\Pi x : [A]. [B])) \ (x : [A]). \]
\[ \text{hbind} \ (\Pi x : [A]. [B]) \ [B] \]
\[ \hat{f} \ (\lambda f : \Pi x : [A]. [B]. f \ x) \]

**Theorem 1 (Computational Soundness).** If \( \Gamma \vdash M \equiv N \) then \([\Gamma]_p \vdash [M]_p \equiv [N]_p\).
The translation

\[
\begin{align*}
\mu \Box_i & := \lambda A : T (T \Box_i). \\
\mu \Pi_{A B} & := \text{allow for a direct implementation in Coq!}
\end{align*}
\]

**Theorem 1** (Computational Soundness). If \( \Gamma \vdash M \equiv N \) then \([\Gamma]_p \vdash [M]_p \equiv [N]_p\).
An Effectful Way to Eliminate Addiction to Dependence

Pierre-Marie Pédrot
University of Ljubljana
pierre-marie.pedrot@fmf.uni-lj.si

Nicolas Tabareau
INRIA
nicolas.tabareau@inria.fr

Abstract—We define a monadic translation of type theory, called weaning translation, that allows for a large range of effects in dependent type theory—such as exceptions, non-termination, non-determinism or writing operation. Through the light of a call-by-push-value decomposition, we explain why the traditional approach fails with type dependency and justify our approach. Crucially, the construction requires that the universe of algebras of the monad forms itself an algebra. The weaning translation applies to a version of the Calculus of Inductive Constructions (CIC) with a restricted version of dependent elimination. Finally, we show how to recover a translation of full CIC by mixing parametricity techniques with the weaning translation. This provides the first effectful version of CIC.

I. INTRODUCTION

The gap between type theories such as CIC and mainstream programming languages comes to a large extend from the absence of effects in type theories, because of its complex interaction with dependency. For instance, it has already been noticed that inductive types and dependent elimination do not scale well to CPS translations and classical logic [1], [2]. Furthermore, the traditional way to integrate effects in functional programming using monads does not scale to dependency because the monad leaks in the type during substitution.

In this paper, we propose Baclofen Type Theory (BTT), a stripped-down version of CIC, and we provide generic notion of syntactic models of it that allows for a large range of effects in dependent type theory—exceptions, non-termination, non-determinism or writing operation. BTT has a restricted version of dependent elimination to overcome the difficulty to marry effect and dependency. The syntactic models are given by the weaning translation of BTT into CIC, using a variant of the traditional monadic translation. The need for this variant can be explained by analyzing the call-by-push-value (CBPV) decomposition of call-by-value and call-by-name reduction strategies. Crucially, our construction requires that the universe of algebras of the monad forms itself an algebra. A monad satisfying this property is said to be self-algebraic. We then show how very common monads satisfy this property and thus give rise to effects that can be integrated to BTT.

Finally, by mixing parametricity techniques with the weaning translation, we show how to recover a translation of full CIC, giving rise to the first effectful version of CIC.

Plan of the paper.

In Section II, we explain through the CBPV decompositions the weaning translation given in this paper. It is based on two key ingredients that allow us to define a monadic translation of CIC for a large range of effects: i) the use of the call-by-name decomposition in Levy’s call-by-push-value [5] (CBPV) and ii) its instantiation with Eilenberg-Moore algebras for self-algebraic monads.

A. The two canonical decompositions of a monad.

The use of monads to interpret effects in functional programming languages comes back from the seminal work of Moggi [6]. In its traditional view, the monadic interpretation amounts to consider functions from A to B as functions of type \( A \to \tau B \) where \( \tau \) is a computational monad. From a categorical point of view, this interpretation consists in working in the Kleisli category \( C_T \) induced by a monad \( T \) on \( C \), where the objects are those of \( C \) and the morphisms of \( C_T(A, B) \) are given by \( C(A, T B) \). Actually, the Kleisli category is one of the two canonical notions of category of computations that is part of a left and right adjoints decomposition of the monad.
Example 1 : Exception
Example 1 : Exception

Demo
Example II: Writer
Example II : Writer

Demo
Example III : Non-Termination
Example III: Non-Termination

Demo
What next?

• Systematize the use of monadic translation in Coq
  Better integration, Univalence?

• Mix CPBV and Dependency
  Develop a language to allow effects in Coq

• Tame effects with parametricity