Certifications of programs with computational effects

PhD Thesis Defense:

Burak Ekici*

Supervisors: Dr. Jean-Guillaume Dumas*, Dr. Dominique Duval*

*LJK, University Joseph Fourier

Committee: Dr. Andrej Bauer
Dr. Catherine Dubois
Dr. Olivier Laurent
Dr. Jean-François Monin
Dr. Damien Pous
Dr. Alan Schmitt

December 9, 2015 - Grenoble, France.

Computational effects

In mathematics;

- an operation (e.g., function) always returns the same result on the same input,
- the result only depends on the input argument(s).

Computational effects

In mathematics;

- an operation (e.g., function) always returns the same result on the same input,
- the result only depends on the input argument(s).

However, in programming;

- a program might do different things than computing the result:
  - fall into an exceptional case, (exceptions)
  - caught by a non-terminating loop, (non-termination)
  - stuck in interaction with the external world (I/O).

Computational effects

In mathematics;

- an operation (e.g., function) always returns the same result on the same input,
- the result only depends on the input argument(s).

However, in programming;

- a program might do different things than computing the result:
  - ⋆ fall into an exceptional case, (exceptions)
  - ⋆ caught by a non-terminating loop, (non-termination)
  - ⋆ stuck in interaction with the external world (I/O).

All such ⋆ phenomena are known as computational effects.

Reasoning about programs involving exceptions...

... is difficult:

- exceptions are computational effects:
  a program $X \to Y$
  is interpreted as a function $X \to Y + E$
  (where E is the set of exceptions)
- the handling mechanism is encapsulated
  in a single `try-catch` block
  which propagates exceptions: $X \to Y + E$
  it relies on the `catch` part
  which recovers from exceptions: $E \to Y + E$

Motivation

Goal: adding features to handle exceptions into a pure language without worsening its (syntactic) completeness.

Motivation

Goal: adding features to handle exceptions into a pure language without worsening its (syntactic) completeness.

Goal (revisited): proving that theories of a decorated logic for exceptions are Hilbert-Post complete with respect some pure sub-logic.

Motivation

Goal: adding features to handle exceptions into a pure language without worsening its (syntactic) completeness.

Goal (revisited): proving that theories of a decorated logic for exceptions are Hilbert-Post complete with respect some pure sub-logic.

Outline:

(1) introduce the decorated logic for exceptions and its theories,

(2) define the relative Hilbert-Post completeness property,

(3) give (a sketch of) a relative Hilbert-Post completeness proof for these decorated theories in a Coq implementation.

Consequence

Thanks to DUALITY between EXCEPTIONS and STATES [Dumas&Duval&Fousse&Reynaud]

we consequently get:

- the decorated logic for states,
- relatively Hilbert-Post complete theories of the decorated logic for states.

Some literature

- About effects:
  - monads [Moggi 1991],
  - effect systems [Lucassen&Gifford 1988],
  - Lawvere theories [Plotkin&Power 2002],
  - algebraic handlers [Plotkin&Pretnar 2009],
  - comonads [Uustalu&Vene 2008] and [Petricek&Orchard&Mycroft 2014],
  - dynamic logic [Mossakowski&Schröder&Goncharov 2010].
- Implementations:
  - Haskell,
  - Eff [Bauer&Pretnar], Idris [Brady].
- About completeness properties of effects:
  - (global) states [Pretnar 2010]
  - local states [Staton 2010].

I.
Decorated logics

Decorated logic

(1) A decorated logic $\mathcal{L}_{dec}$ [Dominguez & Duval'08] is an extension to monadic equational logic $\mathcal{L}_{meq}$ with the use of decorations on terms and equations.

(2) $\mathcal{L}_{dec}$ provides equivalence proofs among `programs with effects`.

Syntax for the monadic equational logic ($\mathcal{L}_{meq}$):

Types:	$\mathtt{t}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \ldots$
Terms:	$\mathtt{f}\ \mathtt{g}$	$::=$	$\mathtt{id}_\mathtt{t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f}$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \cong \mathtt{g}$

Decorated logic

(1) A decorated logic $\mathcal{L}_{dec}$ [Dominguez & Duval'08] is an extension to monadic equational logic $\mathcal{L}_{meq}$ with the use of decorations on terms and equations.

(2) $\mathcal{L}_{dec}$ provides equivalence proofs among `programs with effects`.

Syntax for a decorated logic

Types:	$\mathtt{t}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \ldots$
Terms:	$\mathtt{f}\ \mathtt{g}$	$::=$	$\mathtt{id}_\mathtt{t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f}$
Decoration for terms:	$(\mathtt{d})$	$::=$	$(0) \mid (1) \mid (2)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv \mathtt{g} \mid \mathtt{f} \sim \mathtt{g}$

Decorations are used to classify “effectful” terms.

Decorated logic for exceptions ($\mathcal{L}_{exc}$)

The exceptions effect is handling of exceptions in an imperative programming language.

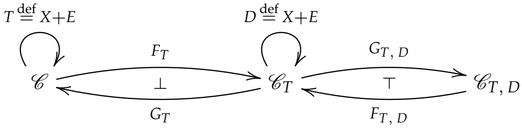
Syntax of the decorated logic for exceptions ($\mathcal{L}_{exc}$): ($e \in EName$)

Types:	$\mathtt{t}\ \mathtt{s}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{t}+\mathtt{s} \mid \mathbb{0} \mid \mathtt{P_e}$
Terms:	$\mathtt{f}\ \mathtt{g}$	$::=$	$\mathtt{id_t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f} \mid [\mathtt{g} \mid \mathtt{f}] \mid$
			$\mathtt{inl} \mid \mathtt{inr} \mid [\,]_\mathtt{t} \mid \mathtt{tag}_e \mid \mathtt{untag}_e \mid \downarrow\mathtt{f}$
Decoration for terms:	$(\mathtt{d})$	$::=$	$(0) \mid (1) \mid (2)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv \mathtt{g} \mid \mathtt{f} \sim \mathtt{g}$

$$\mathtt{tag}_e^{(1)} : \mathtt{P_e} \to \mathbb{0}$$
$$\mathtt{untag}_e^{(2)} : \mathbb{0} \to \mathtt{P_e}$$

Interpreting the logic $\mathcal{L}_{exc}$

The coKleisli-on-Kleisli construction:



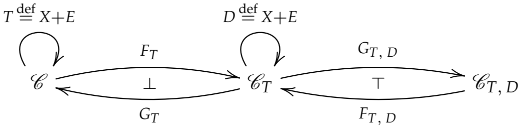
$\eta : Id \Rightarrow T$ $\qquad$ $\varepsilon : D \Rightarrow Id$

Theorem

1. *F_T is faithful.*
2. *the category $\mathscr{C}_{T,D}$ is the full image category of T.*
3. *$G_{T,D}$ is faithful.*

Interpreting the logic $\mathcal{L}_{exc}$

The coKleisli-on-Kleisli construction:



The types are interpreted as the objects of the category $\mathscr{C}$:

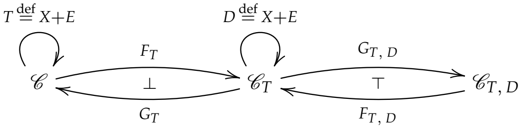
- $\mathbb{0}$ is interpreted as the *initial object*,
- for each e in $EName$, the type P_e is interpreted as an object Par_e,
- the sum type $X + Y$, for each types X and Y, are interpreted as the binary coproducts.

$$E \stackrel{\text{def}}{=} \Sigma_{e \in EName} Par_e$$

with canonical inclusions $in_e : Par_e \to E$.

Interpreting the logic $\mathcal{L}_{exc}$

The coKleisli-on-Kleisli construction:



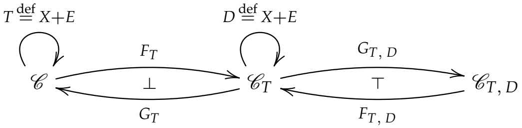
The terms are interpreted as morphisms as follows:

- a *pure* term $f^{(0)} : X \to Y$ in $\mathscr{C}$ as $f : X \to Y$ in $\mathscr{C}$,
- a *propagator* term $f^{(1)} : X \to Y$ in $\mathscr{C}_T$ as $f : X \to Y + E$ in $\mathscr{C}$,
  - $\mathtt{tag}_e^{(1)} : P_e \to \mathbb{0}$ as $\mathtt{tag}_e = in_e : Par_e \to E$
- a *catcher* term $f^{(2)} : X \to Y$ in $\mathscr{C}_{T,D}$ as $f : X + E \to Y + E$ in $\mathscr{C}$
  - $\mathtt{untag}_e^{(2)} : \mathbb{0} \to P_e$ as a term $\mathtt{untag}_e : E \to Par_e + E$ in $\mathscr{C}$ characterized as follows:

$$\begin{cases} \mathtt{untag}_e \circ \mathtt{tag}_e = inl_{Par_e,E} & : Par_e \to Par_e + E \\ \mathtt{untag}_e \circ \mathtt{tag}_f = inr_{Par_e,E} \circ \mathtt{tag}_f & : Par_f \to Par_e + E \quad \text{if } e \neq f \end{cases}$$

Interpreting the logic $\mathcal{L}_{exc}$

The coKleisli-on-Kleisli construction:



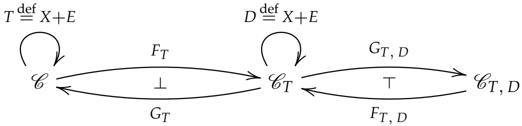
Hierarchy (or conversion) rules among decorations:

$$\frac{f^{(0)}}{f^{(1)}} \quad \text{and} \quad \frac{f^{(1)}}{f^{(2)}}$$

- $\frac{f^{(0)}}{f^{(1)}}$ is interpreted by the functor F_T,
- $\frac{f^{(1)}}{f^{(2)}}$ is interpreted by the functor $G_{T,D}$.
- Consequently $\frac{f^{(0)}}{f^{(2)}}$ is interpreted by the composition $G_{T,D} \circ F_T$.

Interpreting the logic $\mathcal{L}_{exc}$

The coKleisli-on-Kleisli construction:



$\eta : Id \Rightarrow T$ $\qquad\qquad$ $\varepsilon : D \Rightarrow Id$

A strong equation between catchers $f^{(2)} \equiv g^{(2)} : X \to Y$ is interpreted as

$$f = g : X + E \to Y + E \text{ in } \mathscr{C}.$$

A weak equation between catchers $f^{(2)} \sim g^{(2)} : X \to Y$ is interpreted as

$$f \circ \eta_X = g \circ \eta_X : X \to Y + E \text{ in } \mathscr{C}.$$

The fundamental weak equation

- $\texttt{tag}_e^{(1)} \quad : \texttt{P}_\texttt{e} \to \mathbb{0}$
- $\texttt{untag}_e^{(2)} : \mathbb{0} \to \texttt{P}_\texttt{e}$

$$\texttt{untag}_e^{(2)} \circ \texttt{tag}_e^{(1)} \sim id_{P_e}^{(0)}$$

Both members agree on non-exceptional arguments but they may differ on exceptional arguments.

$$\begin{array}{ccccc} & \texttt{tag}_e & & \texttt{untag}_e & \\ p & \mapsto & \boxed{p}_e & \mapsto & p \\ \boxed{p}_e & \mapsto & \boxed{p}_e & \mapsto & p \end{array}$$

Some other rules of $\mathcal{L}_{exc}$

- Conversion rules

$$\frac{f^{(0)}}{f^{(1)}} \qquad \frac{f^{(1)}}{f^{(2)}} \qquad \frac{f^{(d)} \equiv g^{(d')}}{f \sim g} \qquad \frac{f^{(d)} \sim g^{(d')}}{f \equiv g} \text{ if } \mathtt{max}(d, d') \leq 1$$

- The effect rule

$$(\text{effect}) \ \frac{f_1^{(2)}, f_2^{(2)} : X \to Y \qquad f_1^{(2)} \sim f_2^{(2)} \qquad f_1^{(2)} \circ [\,]_X^{(0)} \equiv f_2^{(2)} \circ [\,]_X^{(0)}}{f_1 \equiv f_2}$$

- Decorated versions of the rules of monadic equational logic
- Decorated versions of categorical coproduct rules

$\mathcal{L}_{exc}$ in Coq

Some prerequisites:

```
Parameter EName: Type.
Parameter EVal: EName → Type.
```

The type term is dependent:

```
Inductive term: Type → Type → Type :=
 | comp   : forall {X Y Z: Type}, term X Y → term Y Z → term X Z
 | copair : forall {X Y Z}, term Z X → term Z Y → term Z (X + Y)
 | tpure  : forall {X Y: Type}, (X → Y) → term Y X
 | tag    : e: EName → term Empty_set (EVal e)
 | untag  : e: EName → term (EVal e) Empty_set.
Infix "o" := comp (at level 60).
```

An example:

```
Definition id {X: Type} : term X X := tpure id.
```

Decorations in Coq

Decorations are assigned on terms by a Coq predicate named is:

```
Inductive ekind := epure | ppg | ctc.

Inductive is : ekind → forall X Y, term X Y → Prop :=
 | is_tpure    : forall X Y (f: X → Y), is (epure) (@tpure X Y f)
 | is_comp     : forall k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f o g)
 | is_copair   : forall k X Y Z (f: term Z X) (g: term Z Y), is ppg f → is k f → is k g → is k (copair f g)
 | is_tag      : forall t, is ppg (tag t)
 | is_untag    : forall t, is ctc (untag t)
 | is_epure_ppg : forall X Y k (f: term X Y), is epure f → is ppg f
 | is_ppg_ctc  : forall X Y k (f: term X Y), is ppg f → is ctc f.
Hint Constructors is.
```

Decorations in Coq

Decorations are assigned on terms by a Coq predicate named is:

```
Inductive ekind := epure | ppg | ctc.

Inductive is : ekind → forall X Y, term X Y → Prop :=
 | is_tpure    : forall X Y (f: X → Y), is (epure) (@tpure X Y f)
 | is_comp     : forall k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f o g)
 | is_copair   : forall k X Y Z (f: term Z X) (g: term Z Y), is ppg f → is k f → is k g → is k (copair f g)
 | is_tag      : forall t, is ppg (tag t)
 | is_untag    : forall t, is ctc (untag t)
 | is_epure_ppg : forall X Y k (f: term X Y), is epure f → is ppg f
 | is_ppg_ctc  : forall X Y k (f: term X Y), is ppg f → is ctc f.
Hint Constructors is.
```

A tactic to automatically reason about decorations:

```
Ltac edecorate := solve[repeat
 (apply is_comp || apply is_copair)
                  ||
 (apply is_tpure || apply is_tag || apply is_untag)
                  ||
 (apply is_epure_ppg) || (apply is_ppg_ctc)].
```

Some rules in Coq

The rules are given in a mutually inductive way:

```
Inductive strong: forall X Y, relation (term X Y) :=
        ⋮
 | effect: forall X Y (f g: term Y X), f ∼ g → (f o (empty X) == g o (empty X)) → f == g
 | tcomp: forall X Y Z (f: Z → Y) (g: Y → X), tpure (compose g f) == tpure g o tpure f
with weak: forall X Y, relation (term X Y) :=
        ⋮
 | fundweq: forall e: EName, untag e o tag e ∼ (@id (EVal e))
where "x == y" := (strong x y)
    "x ∼ y" := (weak x y).
```

Programmer's language for exceptions ($\mathcal{L}_{exc-pl}$)

Syntax for the programmer's language: ($e \in EName$)

Types:	$\mathtt{t}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{P_e}$
Terms:	$\mathtt{f}, \mathtt{g}$	$::=$	$\mathtt{id_t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f} \mid$
			$\mathtt{throw}_{t,e} \mid \mathtt{try(f)\,catch(e} \Rightarrow \mathtt{g)}$
Decoration for terms:	$(\mathtt{d})$	$::=$	$(0) \mid (1)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv \mathtt{g}$

Programmer's language for exceptions ($\mathcal{L}_{exc-pl}$)

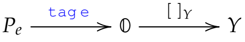
Syntax for the programmer's language: $(e \in EName)$

Types:	$\mathtt{t}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{P_e}$
Terms:	$\mathtt{f},\ \mathtt{g}$	$::=$	$\mathtt{id_t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f} \mid$
			$\mathtt{throw}_{t,\,e} \mid \mathtt{try(f)\,catch(e} \Rightarrow \mathtt{g)}$
Decoration for terms:	$(\mathtt{d})$	$::=$	$(0) \mid (1)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv \mathtt{g}$

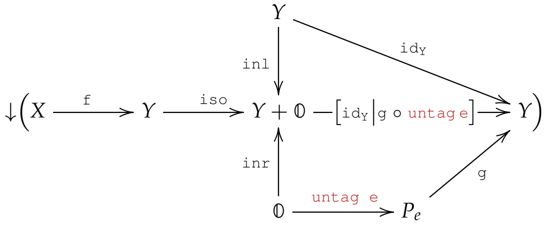
$$\begin{array}{lcl} \mathtt{throw}^{(1)}_{X,\,e} & : & P_e \to X \\ \mathtt{try(a)\,catch(e} \Rightarrow \mathtt{b)}^{(1)} & : & X \to Y \end{array}$$

$$(\text{ppt})\frac{a : X \to Y}{a \circ \mathtt{throw}_{X,\,e} \equiv \mathtt{throw}_{Y,\,e}} \qquad (\text{try}_1)\frac{u^{(0)} : X \to P_e \quad b : P_e \to Y}{\mathtt{try}(\mathtt{throw}_{Y,\,e} \circ u)\mathtt{catch}(e \Rightarrow b) \equiv b \circ u}$$

Translating $\mathcal{L}_{exc-pl}$ into $\mathcal{L}_{exc}$



$\texttt{throw}_{Y,e}$



$\texttt{try}(f)\texttt{catch}(e \Rightarrow g)$

II.
Relative Hilbert-Post Completeness

Categorical view of computation

Various syntactic and semantic notions are treated uniformly

- Syntax: a theory generated by some kind of language (types, terms,...) and equations is a (...)-category
- Semantics: a domain of interpretation is a (...)-category, and a model of a theory in a domain is a (...)-functor

Some examples:
(...)-category = cartesian closed category
for simply typed lambda-calculus

(...)-category = category
for monadic equational logic

(...)-category = decorated category
for the decorated logic for exceptions

An example: monadic equational logic

(...)-category = category
for the logic $\mathcal{L}_{meq,nat}$:

- Syntax: the language $\texttt{Lang}_{meq,nat}$ generated by:

$$\begin{array}{llll} \text{Types:} & \texttt{t} & ::= & \texttt{U} \mid \texttt{N} \\ \text{Terms:} & \texttt{f g} & ::= & \texttt{id}_\texttt{t} \mid \texttt{g} \circ \texttt{f} \mid \texttt{z} \mid \texttt{s} \end{array}$$

  several theories $\mathcal{T}_{meq}$ in $\texttt{Lang}_{meq,nat}$ can be generated by:

$$\text{Equations:} \quad \texttt{e} \quad ::= \quad \{\ldots\}$$

- Semantics: a model of the theory with "no equations" of naturals in $\mathscr{S}et$:

Theory $\rightarrow$	Domain
U	$\{*\}$
N	$\mathbb{N}$
id_t	$x \mapsto x$
z	0
s	$x \mapsto x + 1$

Decorated logic

(...)-category = decorated category
for the logic $\mathcal{L}_{exc-\oplus}$ ($\mathcal{L}_{exc}$ with no "case distinction" and a single exception name):

- Syntax: several languages $\mathtt{Lang}_{exc-\oplus}$ can be generated by:

$$\begin{array}{llll} \text{Types:} & \mathtt{t} & ::= & \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathbb{0} \mid \mathtt{P} \\ \text{Terms:} & \mathtt{f}\ \mathtt{g} & ::= & \mathtt{id}_{\mathtt{t}}^{(0)} \mid [\,]_{\mathtt{t}}^{(0)} \mid \mathtt{a}^{(0)} \mid \mathtt{b}^{(0)} \mid \ldots \mid \mathtt{g}^{(0)} \circ \mathtt{f}^{(0)} \mid \\ & & & \mathtt{tag}^{(1)} \mid \mathtt{untag}^{(2)} \mid \mathtt{g}^{(1)} \circ \mathtt{f}^{(1)} \mid \mathtt{g}^{(2)} \circ \mathtt{f}^{(2)} \end{array}$$

  several theories $\mathcal{T}_{exc}$ in a fixed language $\mathtt{Lang}_{exc-\oplus}$ can be generated by:

$$\text{Equations:} \quad \mathtt{e} \quad ::= \quad \{\ldots^{(0)} \equiv \ldots^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim \mathtt{id}_{\mathtt{P}}^{(0)}\}$$

- Semantics: a model of the theory with "no pure equations" in $\mathscr{S}et$:

Theory $\rightarrow$	Domain	
$\mathbb{0}$	$\{\}$	
$\mathtt{P}$	Par	
$[\,]_t^{(0)}$	empty function	
$\mathtt{tag}^{(1)} : P \rightarrow \mathbb{0}$	$\mathtt{tag} : \mathrm{Par} \rightarrow E$	$p \mapsto \boxed{p}$
$\mathtt{untag}^{(2)} : \mathbb{0} \rightarrow P$	$\mathtt{untag} : E \rightarrow \mathrm{Par} + E$	$\boxed{p} \mapsto p$

Any theory $\mathcal{T}_{exc}$ will be shown as Hilbert-Post complete w.r.t. the logic $\mathcal{L}_{meq}$!

Another example: decorated logic

(...)-category = decorated category for the logic $\mathcal{L}_{exc-\oplus,nat}$

- Syntax: the language $\mathtt{Lang}_{exc-\oplus,nat}$ is generated by:

$$\begin{array}{llll} \text{Types:} & \mathtt{t} & ::= & \mathbb{0} \mid \mathtt{U} \mid \mathtt{N} \\ \text{Terms:} & \mathtt{f}\ \mathtt{g} & ::= & \mathtt{id}_{\mathtt{t}}^{(0)} \mid [\,]_{\mathtt{t}}^{(0)} \mid \mathtt{z}^{(0)} \mid \mathtt{s}^{(0)} \mid \mathtt{g}^{(0)} \circ \mathtt{f}^{(0)} \mid \\ & & & \mathtt{tag}^{(1)} \mid \mathtt{untag}^{(2)} \mid \mathtt{g}^{(1)} \circ \mathtt{f}^{(1)} \mid \mathtt{g}^{(2)} \circ \mathtt{f}^{(2)} \end{array}$$

several theories $\mathcal{T}_{exc,nat}$ in $\mathtt{Lang}_{exc-\oplus,nat}$ is generated by:

$$\text{Equations:} \quad \mathtt{e} \quad ::= \quad \{\ldots^{(0)} \equiv \ldots^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim \mathtt{id}_{\mathtt{N}}^{(0)}\}$$

- Semantics: a model of the theory with "no pure equations" of naturals in $\mathscr{S}et$:

Theory	$\rightarrow$	Domain	
$\mathbb{0}$		$\{\}$	
$\mathtt{U}$		$\{*\}$	
$\mathtt{N}$		$\mathbb{N}$	
$[\,]_t^{(0)}$		empty function	
$\mathtt{tag}^{(1)} : N \rightarrow \mathbb{0}$		$\mathtt{tag} : \mathbb{N} \rightarrow E$	$3 \mapsto \boxed{3}$
$\mathtt{untag}^{(2)} : \mathbb{0} \rightarrow N$		$\mathtt{untag} : \mathbb{N} \rightarrow \mathbb{N} + E$	$\boxed{3} \mapsto 3$

Any theory $\mathcal{T}_{exc,nat}$ will be shown as Hilbert-Post complete w.r.t. the logic $\mathcal{L}_{meq,nat}$!

Soundness and completeness of theories $\mathcal{T}_{exc}$

- In this framework, soundness of the theories $\mathcal{T}_{exc}$ of the logic $\mathcal{L}_{exc-\oplus}$ with respect to denotational semantics is granted:
  Provable $\Longrightarrow$ Valid
- But completeness is not immediate:

* Semantic completeness:
  Valid $\Longrightarrow$ Provable

* Syntactic completeness:

  Every added unprovable sentence introduces an inconsistency, where inconsistency means:

  - either negation inconsistency:
    there is a sentence φ such that φ and $\neg\varphi$ are provable
  - or Hilbert-Post inconsistency:
    every sentence is provable

(Absolute) Hilbert-Post completeness

Definition

Given a logic $\mathcal{L}$ and its maximal theory $\mathcal{T}_{max}$, a theory $\mathcal{T}$ is,

- ⋆ consistent if $\mathcal{T} \neq \mathcal{T}_{max}$,
- ⋆ (absolute) Hilbert-Post complete, if:
  - ⋆⋆ it is consistent
  - ⋆⋆ any theory which contains $\mathcal{T}$ coincides with $\mathcal{T}_{max}$ or with $\mathcal{T}$.

Example: $\mathcal{L}_{meq,nat}$

Types:	t	::=	U | N
Terms:	f g	::=	$\mathtt{id}_\mathtt{t}$ | g ∘ f | z | s

$$
\begin{array}{cc}
\mathcal{T}_{max} & \{s \equiv id_N\} \\
\cup & \\
\mathcal{T}' & \{s \circ 0 \equiv 0,\ s \circ s \equiv s\} \\
\cup & \\
\vdots & \vdots \\
\cup & \\
\mathcal{T}_{mod6} & \{s^6 \equiv id_N\} \\
\cup & \\
\mathcal{T}_{min} & \{\,\}
\end{array}
$$

Example: $\mathcal{L}_{meq,nat}$

Types:	t	::=	U | N
Terms:	f g	::=	id_t | g ∘ f | z | s

		HPC in $\mathcal{L}_{meq,nat}$
$\mathcal{T}_{max}$	$\{s \equiv id_N\}$	
$\cup$		
$\mathcal{T}'$	$\{s \circ 0 \equiv 0,\ s \circ s \equiv s\}$	
$\cup$		
$\vdots$	$\vdots$	
$\cup$		
$\mathcal{T}_{mod6}$	$\{s^6 \equiv id_N\}$	
$\cup$		
$\mathcal{T}_{min}$	$\{\,\}$	X

Example: $\mathcal{L}_{meq,nat}$

Types:	t	::=	U | N
Terms:	f g	::=	$\texttt{id}_\texttt{t}$ | g ∘ f | z | s

		HPC in $\mathcal{L}_{meq,nat}$
$\mathcal{T}_{max}$	$\{s \equiv id_N\}$	
$\cup$		
$\mathcal{T}'$	$\{s \circ 0 \equiv 0,\ s \circ s \equiv s\}$	
$\cup$		
$\vdots$	$\vdots$	
$\cup$		
$\mathcal{T}_{mod6}$	$\{s^6 \equiv id_N\}$	X
$\cup$		
$\mathcal{T}_{min}$	$\{\}$	X

Example: $\mathcal{L}_{meq,nat}$

Types:	t	::=	U | N
Terms:	f g	::=	id_t | g ∘ f | z | s

		HPC in $\mathcal{L}_{meq,nat}$
$\mathcal{T}_{max}$	$\{s \equiv id_N\}$	
$\cup$		
$\mathcal{T}'$	$\{s \circ 0 \equiv 0,\ s \circ s \equiv s\}$	$\surd$
$\cup$		
$\vdots$	$\vdots$	
$\cup$		
$\mathcal{T}_{mod6}$	$\{s^6 \equiv id_N\}$	X
$\cup$		
$\mathcal{T}_{min}$	$\{\,\}$	X

Example: $\mathcal{L}_{meq,nat}$

Types:	`t`	::=	`U` | `N`
Terms:	`f g`	::=	`id`$_{\texttt{t}}$ | `g` ∘ `f` | `z` | `s`

		HPC in $\mathcal{L}_{meq,nat}$
$\mathcal{T}_{max}$	$\{s \equiv id_N\}$	X
$\cup$		
$\mathcal{T}'$	$\{s \circ 0 \equiv 0,\ s \circ s \equiv s\}$	$\surd$
$\cup$		
$\vdots$	$\vdots$	
$\cup$		
$\mathcal{T}_{mod6}$	$\{s^6 \equiv id_N\}$	X
$\cup$		
$\mathcal{T}_{min}$	$\{\,\}$	X

Relative Hilbert-Post completeness

Definition

$$\mathrm{Theory}(\mathcal{L}_0) \underset{G}{\overset{F}{\rightleftarrows}} \mathrm{Theory}(\mathcal{L}) \quad (F \dashv G,\ \bot)$$

A theory $\mathcal{T}$ of $\mathcal{L}$ is Hilbert-Post complete with respect to $\mathcal{L}_0$ if

- ⋆⋆ it is consistent and
- ⋆⋆ each formula e of $\mathcal{L}$ is $\mathcal{T}$-equivalent to some set E_0 of formulae of the logic $\mathcal{L}_0$:

$$\mathcal{T} + Th(e) = \mathcal{T} + Th(E_0)$$

The *relative Hilbert-Post completeness* lifts the *absolute* one from the logic $\mathcal{L}_0$ to the logic $\mathcal{L}$.

Example: $\mathcal{L}_{meq,nat}$ and $\mathcal{L}_{exc-\oplus,nat}$

$$\text{Theory}(\mathcal{L}_{meq,nat}) \underset{G}{\overset{F}{\rightleftarrows}} \text{Theory}(\mathcal{L}_{exc-\oplus,nat}) \quad (F \dashv G)$$

Types:	t	::=	$\mathbb{0}$ | U | N
Terms:	f g	::=	$\mathtt{id}_t^{(0)}$ | $[\,]_t^{(0)}$ | $\mathtt{z}^{(0)}$ | $\mathtt{s}^{(0)}$ | $\mathtt{g}^{(0)} \circ \mathtt{f}^{(0)}$ | $\mathtt{tag}^{(1)}$ | $\mathtt{untag}^{(2)}$ | $\mathtt{g}^{(1)} \circ \mathtt{f}^{(1)}$ | $\mathtt{g}^{(2)} \circ \mathtt{f}^{(2)}$

HPC in $\mathcal{L}_{exc-\oplus,nat}$

$F(\mathcal{T}_{max})$ $\quad \{s^{(0)} \equiv id_N^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$

$\cup$

$F(\mathcal{T}')$ $\quad \{s^{(0)} \circ 0^{(0)} \equiv 0^{(0)},\ s^{(0)} \circ s^{(0)} \equiv s^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$ $\quad$?

$\cup$

$\vdots \qquad \vdots$

$\cup$

$F(\mathcal{T}_{mod6})$ $\quad \{s^{6(0)} \equiv id_N^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$

$\cup$

$F(\mathcal{T}_{min})$ $\quad \{\mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$

Example: $\mathcal{L}_{meq,nat}$ and $\mathcal{L}_{exc-\oplus,nat}$

$$\text{Theory}(\mathcal{L}_{meq,nat}) \underset{G}{\overset{F}{\rightleftarrows}} \text{Theory}(\mathcal{L}_{exc-\oplus,nat}) \quad (F \dashv G)$$

Types:	t	::=	$\mathbb{0}$ | U | N
Terms:	f g	::=	$\mathtt{id}_t^{(0)}$ | $[\,]_t^{(0)}$ | $\mathtt{z}^{(0)}$ | $\mathtt{s}^{(0)}$ | $\mathtt{g}^{(0)} \circ \mathtt{f}^{(0)}$ |
			$\mathtt{tag}^{(1)}$ | $\mathtt{untag}^{(2)}$ | $\mathtt{g}^{(1)} \circ \mathtt{f}^{(1)}$ | $\mathtt{g}^{(2)} \circ \mathtt{f}^{(2)}$

HPC in $\mathcal{L}_{exc-\oplus,nat}$

$F(\mathcal{T}_{max})$ $\quad \{s^{(0)} \equiv id_N^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$

$\cup$

$F(\mathcal{T}')$ $\quad \{s^{(0)} \circ 0^{(0)} \equiv 0^{(0)},\ s^{(0)} \circ s^{(0)} \equiv s^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$ $\quad \checkmark$

$\cup$

$\vdots \qquad \vdots$

$\cup$

$F(\mathcal{T}_{mod6})$ $\quad \{s^{6(0)} \equiv id_N^{(0)},\ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$

$\cup$

$F(\mathcal{T}_{min})$ $\quad \{\mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \sim id_N^{(0)}\}$

III.
Relative Hilbert-Post Completeness in Coq

The proof sketch

Thanks to the relative Hilbert-Post completeness definition, we get:

Goal: proving that for each equation e in $\mathcal{L}_{exc-\oplus}$ is $\mathcal{T}_{exc}$-equivalent to a finite set E_0 of equations in the pure logic $\mathcal{L}_{meq}$.

The proof sketch:

(1) decide the canonical forms for propagators and catchers,

(2) show that any equation e (made of canonical forms) in $\mathcal{L}_{exc-\oplus}$ is T_{exc}-equivalent to a finite set of equations in the pure sub-logic $\mathcal{L}_{meq}$.

Restriction on the use of copairs/coproducts:
it is easier to determine the canonical forms of propagator and catchers in the absence of categorical copairs/coproducts.

⇒To be considered...

Canonical forms

Proposition

- *For each propagator* $a^{(1)} : X \to Y$*, either* a *is pure or there is a pure term* $v^{(0)} : X \to P$ *such that*

$$a^{(1)} \equiv [\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v^{(0)}.$$

- *For each catcher* $f^{(2)} : X \to Y$*, either* f *is a propagator or there is a propagator* $a^{(1)} : P \to Y$ *and a pure term* $u^{(0)} : X \to P$ *such that*

$$f^{(2)} \equiv a^{(1)} \circ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \circ v^{(0)}.$$

Canonical forms in Coq

```
(** Canonical form for propagators **)
Lemma can_propagators: forall {X Y} (a: term Y X), has_no_catcher a →
   (has_only_pure a
     ∨
   (exists v :(term (Val e) X),
   (has_only_pure v) ∧ (a == ((@empty Y) o tag e o v)))).

(** Canonical form for catchers **)
Lemma can_catchers: forall {X Y} (f: term Y X),
   (has_no_catcher f
     ∨
   (exists a: (term Y (Val e)), exists u: (term (Val e) X),
   (has_no_catcher a) ∧ (has_only_pure u) ∧ (f == (a o untag e o tag e o u)))).
```

Canonical forms in Coq

```
(** Canonical form for propagators **)
Lemma can_propagators: forall {X Y} (a: term Y X), has_no_catcher a →
   (has_only_pure a
     ∨
   (exists v :(term (Val e) X),
   (has_only_pure v) ∧ (a == ((@empty Y) ∘ tag e ∘ v)))).

(** Canonical form for catchers **)
Lemma can_catchers: forall {X Y} (f: term Y X),
   (has_no_catcher f
     ∨
   (exists a: (term Y (Val e)), exists u: (term (Val e) X),
   (has_no_catcher a) ∧ (has_only_pure u) ∧ (f == (a ∘ untag e ∘ tag e ∘ u)))).
```

Key point: benefiting the structural induction!

Equivalences between terms

Lemma[a]

An equation between propagators is $\mathcal{T}_{exc}$-equivalent to a set of equations between pure terms.

Equivalences between terms

Lemma[a]

An equation between propagators is $\mathcal{T}_{exc}$-equivalent to a set of equations between pure terms.

Lemma

An equation between catchers is $\mathcal{T}_{exc}$-equivalent to a set of equations between propagators.

Equivalences between terms

Lemma[a]

An equation between propagators is $\mathcal{T}_{exc}$-equivalent to a set of equations between pure terms.

Lemma

An equation between catchers is $\mathcal{T}_{exc}$-equivalent to a set of equations between propagators.

Theorem[a]

Any theory $\mathcal{T}_{exc}$ of the logic $\mathcal{L}_{exc-\oplus}$ is relatively Hilbert-Post complete with respect to the pure logic $\mathcal{L}_{meq}$.

[a] Under some technical assumption.

Main theorem in Coq

```
(** An equation between any two terms is either absurd or
T_exc-equivalent to two equations between pure terms. **)
Theorem Theorem_6_10_9: forall {X Y} (f1 f2: term Y X), (Val e <> empty_set) →
        ((( f1 == f2) ↔ (forall {X Y} (f g: term Y X), f == g))
         ∨
        (exists a1: (term Y X), exists a2: (term Y X),
         exists b1: (term (Val e) (Val e)), exists b2: (term (Val e) (Val e)),
         (has_only_pure a1 ∧ has_only_pure a2 ∧
         has_only_pure b1 ∧ has_only_pure b2 ∧
         (f1 == f2 ↔ (a1 == a2 ∧ b1 == b2))))
         ∨
        (exists a1: (term (Val e) X), exists a2: (term (Val e) X),
         exists b1: (term (Val e) (Val e)), exists b2: (term (Val e) (Val e)),
         (has_only_pure a1 ∧ has_only_pure a2 ∧
         has_only_pure b1 ∧ has_only_pure b2 ∧
         (f1 == f2 ↔ (a1 == a2 ∧ b1 == b2))))
         ∨
        (exists a1: (term (Val e) X), exists a2: (term (Val e) X),
         exists b1: (term Y (Val e)), exists b2: (term Y (Val e)),
         (has_only_pure a1 ∧ has_only_pure a2 ∧
         has_only_pure b1 ∧ has_only_pure b2 ∧
         (f1 == f2 ↔ (a1 == a2 ∧ b1 == b2))))
         ∨
        (exists a1: (term Y X), exists a2: (term Y X),
         exists b1: (term Y (Val e)), exists b2: (term Y (Val e)),
         (has_only_pure a1 ∧ has_only_pure a2 ∧
         has_only_pure b1 ∧ has_only_pure b2 ∧
         (f1 == f2 ↔ (a1 == a2 ∧ b1 == b2))))
).
```

Summary

We have introduced;

- the logics $\mathcal{L}_{meq}$, $\mathcal{L}_{exc}$ and $\mathcal{L}_{exc-\oplus}$,
- theories $\mathcal{T}_{exc}$ of the logic $\mathcal{L}_{exc-\oplus}$.

We have defined the relative Hilbert-Post completeness property.

We have proven that theories $\mathcal{T}_{exc}$ of $\mathcal{L}_{exc-\oplus}$ is relatively Hilbert-Post complete.

Perspectives

(1) checking whether the theory $\mathcal{T}_{exc}$ of the logic $\mathcal{L}_{exc}$ is relatively Hilbert-Post complete:

- several exception names
- case distinction

(2) an application of "decorated equational reasoning" to an imperative language:

- first attempt: equivalence proofs between programs (mixing states and exceptions) written in `IMPEX`

  ⋆ Coq library: https://forge.imag.fr/frs/download.php/697/IMPEX-STATES-EXCEPTIONS-THESIS.tar.gz

(3) combining effects?

An example: IMPEX

E.g.,

```
prog_1 = (
  var x, y ;
  x := 1 ; y := 20 ;
  try(
    while(tt) do (
      if(x <= 0)
      then(throw e)
      else(x := x - 1)
    )
  )
  catch e => (y := 7) ;
) .
```

===

```
prog_2 = (
 var x, y ;
 x := 0 ; y := 7 ;
) .
```

```
Lemma lemma3: forall (x y: Loc), forall (e: EName), x <> y ->
      {{x ::= (const 1) ;;
        (y ::= (const 20)) ;;
        TRY(WHILE (const true)
              DO(IFB ((loc x) <<= (const 0))
                   THEN (THROW e)
                 ELSE(x ::= ((loc x) +++ (const (-1))))
                 ENDIF)
             ENDWHILE)
        CATCH e => (y ::= (const 7))}}
      ===
      {{x ::= (const 0) ;;
        (y ::= (const 7))}}.
Proof.
   intros. simpl. unfold try_catch.
     apply (@swtoss _ _ rw). apply is_comp. apply is_ro_rw, is_pure_ro, is_downcast.
     edecorate. edecorate.
(*tackling downcast*)
     transitivity( ((copair id ((update y o constant 7) o untag e) o coproj1)
```

```
1 subgoals
x : Loc
y : Loc
e : EName
H : x <> y
______________________________________(1/1)
downcast
  ((copair id ((update y o constant 7) o untag e) o coproj1)
   o (copair
        (lpi (pbl o constant true)
           (copair (throw () e)
              (update x o (tpure add o pair (lookup x) (constant (-1))))
             o (pbl o (tpure chkle o pair (lookup x) (constant 0))))
          o (copair (throw () e)
              (update x o (tpure add o pair (lookup x) (constant (-1))))
             o (pbl o (tpure chkle o pair (lookup x) (constant 0))))) id
      o (pbl o constant true)))
o ((update y o constant 20) o (update x o constant 1)) ===
(update y o constant 7) o (update x o constant 0)
```

The end!

Many thanks for your kind attention!

Questions?

IV.
Appendices

Decorated logic for the global state ($\mathcal{L}_{st}$)

The global state effect is handling memory locations in an imperative programming language.

Syntax of the decorated logic for states ($\mathcal{L}_{st}$): $(i \in Loc)$

Types:	$\mathtt{t}\ \mathtt{s}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{t} \times \mathtt{s} \mid \mathbb{1} \mid \mathtt{V_i}$
Terms:	$\mathtt{f}\ \mathtt{g}$	$::=$	$\mathtt{id_t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f} \mid \langle \mathtt{g}, \mathtt{f} \rangle \mid$
			$\pi_1 \mid \pi_2 \mid \langle\ \rangle_{\mathtt{t}} \mid \mathtt{lookup}_i \mid \mathtt{update}_i$
Decoration for terms:	$(\mathtt{d})$	$::=$	$(0) \mid (1) \mid (2)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv \mathtt{g} \mid \mathtt{f} \sim \mathtt{g}$

$$\begin{aligned} \mathtt{lookup}_i^{(1)} &: \mathbb{1} \to \mathtt{V_i} \\ \mathtt{update}_i^{(2)} &: \mathtt{V_i} \to \mathbb{1} \end{aligned}$$

The decorated logic: the states & the exceptions

The combined decorated logic for the state and the exception: $\mathcal{L}_{st+exc}$.

Grammar of the decorated logic for the state + the exception:

Types:	$\mathtt{t}$	$::=$	$\mathtt{merged}$
Terms:	$\mathtt{f}\ \mathtt{g}$	$::=$	$\mathtt{merged}$
Decoration for terms:	$(\mathtt{d}^{\mathtt{s}}, \mathtt{d}^{\mathtt{e}})$	$::=$	$(0^s,0^e) \mid (0^s,1^e) \mid (0^s,2^e) \mid (1^s,0^e) \mid (1^s,1^e) \mid$ $(1^s,2^e) \mid (2^s,0^e) \mid (2^s,1^e) \mid (2^s,2^e)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv\equiv \mathtt{g} \mid \mathtt{f} \equiv\sim \mathtt{g} \mid \mathtt{f} \sim\equiv \mathtt{g} \mid \mathtt{f} \sim\sim \mathtt{g}$

The decorated logic: the states & the exceptions

The combined decorated logic for the state and the exception: $\mathcal{L}_{st+exc}$.

Grammar of the decorated logic for the state + the exception:

Types:	$\mathtt{t}$	$::=$	$\mathtt{merged}$
Terms:	$\mathtt{f}\ \mathtt{g}$	$::=$	$\mathtt{merged}$
Decoration for terms:	$(\mathtt{d}^{\mathtt{s}}, \mathtt{d}^{\mathtt{e}})$	$::=$	$(0^s, 0^e) \mid (0^s, 1^e) \mid (0^s, 2^e) \mid (1^s, 0^e) \mid (1^s, 1^e) \mid$
			$(1^s, 2^e) \mid (2^s, 0^e) \mid (2^s, 1^e) \mid (2^s, 2^e)$
Equations:	$\mathtt{e}$	$::=$	$\mathtt{f} \equiv\equiv \mathtt{g} \mid \mathtt{f} \equiv\sim \mathtt{g} \mid \mathtt{f} \sim\equiv \mathtt{g} \mid \mathtt{f} \sim\sim \mathtt{g}$

Rules are combined.

The state + the exception: terms in Coq

Some prerequisites:

```
Parameter Loc: Type.
Parameter Val: Loc → Type.
Parameter EName: Type.
Parameter EVal: EName → Type.
```

The type `term` is dependent:

```
Inductive term: Type → Type → Type :=
 | comp    : forall {X Y Z: Type}, term X Y → term Y Z → term X Z
 | pair    : forall {X Y Z: Type}, term X Z → term Y Z → term (X*Y) Z
 | copair  : forall {X Y Z: Type}, term Z X → term Z Y → term Z (X + Y)
 | tpure   : forall {X Y: Type}, (X → Y) → term Y X
 | lookup  : forall i:Loc, term (Val i) unit
 | update  : forall i:Loc, term unit (Val i)
 | tag     : forall e:EName, term Empty_set (EVal e)
 | untag   : forall e:EName, term (EVal e) Empty_set.
Infix "o" := comp (at level 60).
```

The state + the exception: terms in Coq

Some prerequisites:

```
Parameter Loc: Type.
Parameter Val: Loc → Type.
Parameter EName: Type.
Parameter EVal: EName → Type.
```

The type `term` is dependent:

```
Inductive term: Type → Type → Type :=
 | comp    : forall {X Y Z: Type}, term X Y → term Y Z → term X Z
 | pair    : forall {X Y Z: Type}, term X Z → term Y Z → term (X*Y) Z
 | copair  : forall {X Y Z: Type}, term Z X → term Z Y → term Z (X + Y)
 | tpure   : forall {X Y: Type}, (X → Y) → term Y X
 | lookup  : forall i:Loc, term (Val i) unit
 | update  : forall i:Loc, term unit (Val i)
 | tag     : forall e:EName, term Empty_set (EVal e)
 | untag   : forall e:EName, term (EVal e) Empty_set.
Infix "o" := comp (at level 60).
```

An example:

```
Definition id {X: Type} : term X X := tpure id.
```

The state + the exception: decorations in Coq

Thereby, the decorations' implementation follows:

```
Inductive kind := pure | ro | rw.
Inductive ekind := epure | ppg | ctc.

Inductive is : ((kind * ekind)%type) → forall X Y, term X Y → Prop :=
  | is_tpure    : forall X Y (f: X → Y), is (pure, epure) (@tpure X Y f)
  | is_comp     : forall k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f o g)
  | is_pair     : forall k k1 X Y Z (f: term X Z) (g: term Y Z), is (ro, k1) f → is k f → is k g → is k (pair f g)
  | is_copair   : forall k k1 X Y Z (f: term Z X) (g: term Z Y), is (k1, ppg) f → is k f → is k g → is k (copair f g)
  | is_lookup   : forall i, is (ro, epure) (lookup i)
  | is_update   : forall i, is (rw, epure) (update i)
  | is_tag      : forall t, is (pure, ppg) (tag t)
  | is_untag    : forall t, is (pure, ctc) (untag t)
  | is_pure_ro  : forall X Y k (f: term X Y), is (pure, k) f → is (ro, k) f
  | is_ro_rw    : forall X Y k (f: term X Y), is (ro, k) f → is (rw, k) f
  | is_pure_ppg : forall X Y k (f: term X Y), is (k, epure) f → is (k, ppg) f
  | is_ppg_ctc  : forall X Y k (f: term X Y), is (k, ppg) f → is (k, ctc) f.
Hint Constructors is.
```

The state + the exception: decorations in Coq

Thereby, the decorations' implementation follows:

```
Inductive kind := pure | ro | rw.
Inductive ekind := epure | ppg | ctc.

Inductive is : ((kind * ekind)%type) → forall X Y, term X Y → Prop :=
  | is_tpure    : forall X Y (f: X → Y), is (pure, epure) (@tpure X Y f)
  | is_comp     : forall k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f o g)
  | is_pair     : forall k k1 X Y Z (f: term X Z) (g: term Y Z), is (ro, k1) f → is k f → is k g → is k (pair f g)
  | is_copair   : forall k k1 X Y Z (f: term Z X) (g: term Z Y), is (k1, ppg) f → is k f → is k g → is k (copair f g)
  | is_lookup   : forall i, is (ro, epure) (lookup i)
  | is_update   : forall i, is (rw, epure) (update i)
  | is_tag      : forall t, is (pure, ppg) (tag t)
  | is_untag    : forall t, is (pure, ctc) (untag t)
  | is_pure_ro  : forall X Y k (f: term X Y), is (pure, k) f → is (ro, k) f
  | is_ro_rw    : forall X Y k (f: term X Y), is (ro, k) f → is (rw, k) f
  | is_pure_ppg : forall X Y k (f: term X Y), is (k, epure) f → is (k, ppg) f
  | is_ppg_ctc  : forall X Y k (f: term X Y), is (k, ppg) f → is (k, ctc) f.
Hint Constructors is.
```

A tactic to automatically reason about decorations:

```
Ltac decorate := solve[repeat
 (apply is_comp || apply is_pair || apply is_copair)
                          ||
 (apply is_tpure || apply is_lookup || apply is_update || apply is_tag || apply is_untag)
                          ||
 (apply is_pure_ro) || (apply is_ro_rw) || (apply is_pure_ppg) || (apply is_pure_ctc)].
```

The state + the exception: some rules in Coq

⇒ The rules are given in a mutually inductive way:

```
Inductive ss: forall X Y, relation (term X Y) :=
 | eq1: forall X Y k (f g: term X Y), RO k f → RO k g → f ∼== g → f === g
 | effect: forall X Y (f g: term Y X), forget o f === forget o g → f ∼== g → f === g
 | eeffect: forall X Y (f g: term Y X), f ==∼ g → (f o (empty X) === g o (empty X)) → f === g
with ws: forall X Y, relation (term X Y) :=
 | eeq1: forall X Y k (f g: term X Y), PPG k f → PPG k g → f ==∼ g → f === g
 | ax1: forall i, lookup i o update i ∼== (@id (Val i))
with sw: forall X Y, relation (term X Y) :=
 | eax1: forall t: EName, untag t o tag t ==∼ (@id unit)
with ww: forall X Y, relation (term X Y) :=
   ...
where "x  ===  y" := (ss x y) and "x  ∼==  y" := (ws x y) and
      "x  ==∼  y" := (sw x y) and "x  ∼∼  y" := (ww x y).
```

IMPEX

`IMPEX` is an imperative language with abilities to handle exceptional cases:

IMPEX

`IMPEX` is an imperative language with abilities to handle exceptional cases:

`Syntax:`

$$\begin{array}{llll} \texttt{aexp:} & \texttt{a}_1\ \texttt{a}_2 & ::= & \texttt{...} \\ \texttt{bexp:} & \texttt{b}_1\ \texttt{b}_2 & ::= & \texttt{...} \\ \texttt{cmd :} & \texttt{c}_1\ \texttt{c}_2 & ::= & \texttt{skip} \mid \texttt{x := a} \mid \texttt{c}_1;\texttt{c}_2 \mid \texttt{if b then c}_1\ \texttt{else c}_2 \mid \\ & & & \texttt{while b do c}_1 \mid \texttt{throw e} \mid \texttt{try c}_1\ \texttt{catch e} \Rightarrow \texttt{c}_2 \end{array}$$

IMPEX

IMPEX is an imperative language with abilities to handle exceptional cases:

Syntax:

aexp: $a_1\ a_2$ ::= ...

bexp: $b_1\ b_2$ ::= ...

cmd : $c_1\ c_2$ ::= skip | x := a | $c_1; c_2$ | if b then c_1 else c_2 |
while b do c_1 | throw e | try c_1 catch e ⇒ c_2

We design equational semantics of IMPEX over combined decorated logic.

IMPEX over decorated logic: Coq implementation

Commands:

```
Inductive Cmd : Type :=
  | skip     : Cmd
  | sequence : Cmd      → Cmd → Cmd
  | assign   : Loc      → Exp Z → Cmd
  | cond     : Exp bool → Cmd → Cmd → Cmd
  | while    : Exp bool → Cmd → Cmd
  | THROW    : EName    → Cmd
  | TRY_CATCH : EName → Cmd → Cmd → Cmd.
```

Translating commands into decorated settings:

```
Fixpoint dCmd (c: Cmd): (term unit unit) :=
 match c with
  | skip              ⇒ (@id unit)
  | sequence c0 c1    ⇒ (dCmd c1) ∘ (dCmd c0)
  | assign j e0       ⇒ (update j) ∘ (dExp e0)
  | cond b c2 c3      ⇒ copair (dCmd c2) (dCmd c3) ∘ (prop2bool ∘ (dExp b))
  | while b c4        ⇒ (copair (loopiter (prop2bool ∘ (dExp b)) (dCmd c4) ∘
                                (dCmd c4)) (@id unit)) ∘ (prop2bool ∘ (dExp b))
  | THROW e           ⇒ (throw unit e)
  | TRY_CATCH e c1 c2 ⇒ (@try_catch _ _ e (dCmd c1) (dCmd c2))
 end.
```

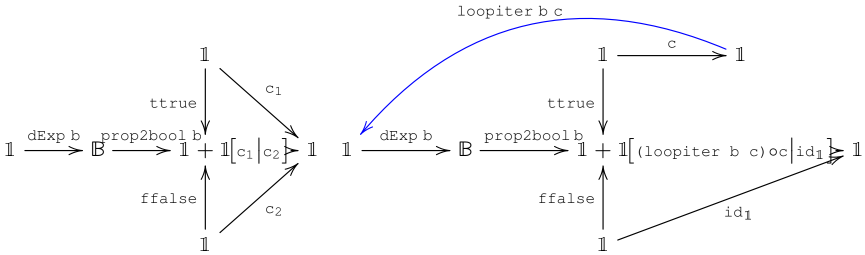


Figure: (if b then c_1 else c_2) and (while b do c) in decorated settings

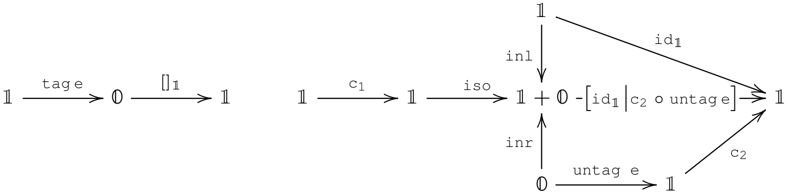


Figure: (throw e) and (try c_1 catch e ⇒ c_2) in decorated settings

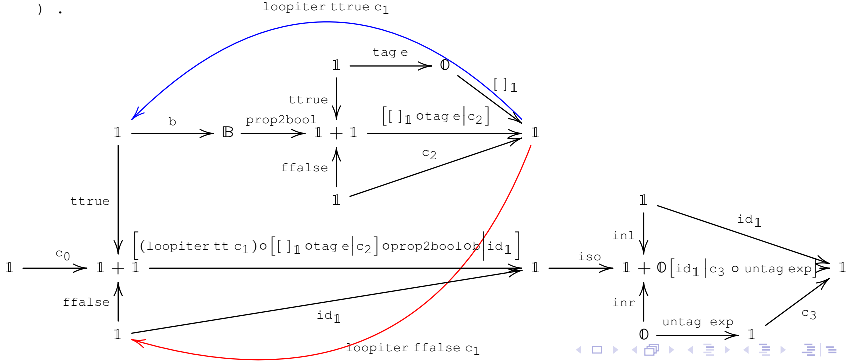
Soundness of the implementation

E.g.,

```
prog_1 = (
  var x, y ;
  x := 1 ; y := 20 ; //c_0
  try(
    while(tt) do (
      if(x <= 0) // c_1
      then(throw e)
      else(x := x - 1) //c_2
    )
  )
  catch e => (y := 7) //c_3 ;
) .
```

===

```
prog_2 = (
 var x, y ;
 x := 0 ; y := 7 ;
) .
```



Soundness of the implementation

E.g.,

```
prog_1 = (
  var x, y ;
  x := 1 ; y := 20 ; //c_0
  try(
    while(tt) do (
      if(x <= 0) // c_1
      then(throw e)
      else(x := x - 1) //c_2
    )
  )
  catch e => (y := 7) //c_3 ;
) .
```

===

```
prog_2 = (
 var x, y ;
 x := 0 ; y := 7 ;
) .
```

$$\mathbb{1} \xrightarrow{\texttt{const 0}} \mathbb{Z} \xrightarrow{\texttt{update x}} \mathbb{1} \xrightarrow{\texttt{const 7}} \mathbb{Z} \xrightarrow{\texttt{update y}} \mathbb{1}$$

Proof verification

E.g.,

```
prog_1 = (
  var x, y ;
  x := 1 ; y := 20 ; //c_0
  try(
    while(tt) do (
      if(x <= 0) // c_1
      then(throw e)
      else(x := x - 1) //c_2
    )
  )
  catch e => (y := 7) //c_3 ;
) .
```

===

```
prog_2 = (
 var x, y ;
 x := 0 ; y := 7 ;
) .
```

```
Lemma lemma3: forall (x y: Loc), forall (e: EName), x <> y ->
      {{x ::= (const 1) ;;
        (y ::= (const 20)) ;;
        TRY(WHILE (const true)
              DO(IFB ((loc x) <<= (const 0))
                   THEN (THROW e)
                 ELSE(x ::= ((loc x) +++ (const (-1))))
                 ENDIF)
             ENDWHILE)
        CATCH e => (y ::= (const 7))}}
      ===
      {{x ::= (const 0) ;;
        (y ::= (const 7))}}.
Proof.
   intros. simpl. unfold try_catch.
     apply (@swtoss _ _ rw). apply is_comp. apply is_ro_rw, is_pure_ro, is_downcast.
     edecorate. edecorate.
(*tackling downcast*)
     transitivity( ((copair id ((update y o constant 7) o untag e) o coproj1)
```

```
1 subgoals
x : Loc
y : Loc
e : EName
H : x <> y
______________________________________(1/1)
downcast
  ((copair id ((update y o constant 7) o untag e) o coproj1)
   o (copair
        (lpi (pbl o constant true)
           (copair (throw () e)
              (update x o (tpure add o pair (lookup x) (constant (-1))))
            o (pbl o (tpure chkle o pair (lookup x) (constant 0))))
         o (copair (throw () e)
              (update x o (tpure add o pair (lookup x) (constant (-1))))
            o (pbl o (tpure chkle o pair (lookup x) (constant 0))))) id
      o (pbl o constant true)))
o ((update y o constant 20) o (update x o constant 1)) ===
(update y o constant 7) o (update x o constant 0)
```

A sketch of the proof

E.g.,

```
prog_1 = (
  var x, y ;
  x := 1 ; y := 20 ; //c_0
  try(
    while(tt) do (
      if(x <= 0) // c_1
      then(throw e)
      else(x := x - 1) //c_2
    )
  )
  catch e => (y := 7) //c_3 ;
) .
```

===

```
prog_2 = (
 var x, y ;
 x := 0 ; y := 7 ;
) .
```

Some bench info:

(1) proof text size is 7.2K

(2) proof verification takes 5.974s with

  (2.1) The Coq Proof Assistant, version 8.4pl3 (January 2014)
  (2.2) Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz

A sketch of the proof:

(1) deal with the first loop iteration which has the state but no exception effect.

(2) study the second iteration of the loop where an exception is thrown.

(3) explain the loop termination followed by the exception recovery and the program termination.

Minimal/maximal theories of a logic

Given a logic $\mathcal{L}$:

- the theories $\mathcal{T}$ of $\mathcal{L}$ are partially ordered by inclusion ($\subseteq$),
- there is a maximal theory $\mathcal{T}_{max}$ of $\mathcal{L}$ where all formulae are theorems,
- there is a minimal theory $\mathcal{T}_{min}$ of $\mathcal{L}$ which is generated by the *empty set* of equations.

Notation: $\mathcal{T} + \mathcal{T}'$ denotes the theory generated by $\mathcal{T}$ and $\mathcal{T}'$.

Minimal/maximal theories of a logic (cont'd)

In an *equational logic*;

- formulae are pairs of parallel terms $(f, g)\colon X \to Y$,
- theorems are equations $f \equiv g\colon X \to Y$.

The *language* of any equational logic may be defined from a *signature* made of sorts and operations.

The *deduction rules* are such that equations form a *congruence*.
I.e., an *equivalence relation* compatible with the term structure.

Minimal/maximal theories of a logic (cont'd)

In an *equational logic*;

- formulae are pairs of parallel terms $(f, g) \colon X \to Y$,
- theorems are equations $f \equiv g \colon X \to Y$.

The *language* of any equational logic may be defined from a *signature* made of sorts and operations.

The *deduction rules* are such that equations form a *congruence*.
I.e., an *equivalence relation* compatible with the term structure.

Example

Consider the logic of naturals $\mathcal{L}_{nat}$ with a language made of
sorts (t) $:= \{*\}, \mathbb{N}$ and
operations $:= id_t \colon t \to t,\ 0 \colon \{*\} \to \mathbb{N}$ and $s \colon \mathbb{N} \to \mathbb{N}$.
Then;

- the minimal theory $\mathcal{T}_{min}$ is generated by *empty set* of equations,
- the maximal theory $\mathcal{T}_{max}$ is generated by $\{s \equiv id_{\mathbb{N}}\}$.

Extensions of a logic

If a logic $\mathcal{L}$ is an extension of a sublogic $\mathcal{L}_0$, then:

(1) each theory $\mathcal{T}_0$ of $\mathcal{L}_0$ generates a theory $F(\mathcal{T}_0)$ of $\mathcal{L}$,

(2) each theory $\mathcal{T}$ of $\mathcal{L}$ determines a theory $G(\mathcal{T})$ of $\mathcal{L}_0$ made of theorems of $\mathcal{T}$ which are formulae of $\mathcal{L}_0$.

Extensions of a logic

If a logic $\mathcal{L}$ is an extension of a sublogic $\mathcal{L}_0$, then:

(1) each theory $\mathcal{T}_0$ of $\mathcal{L}_0$ generates a theory $F(\mathcal{T}_0)$ of $\mathcal{L}$,

(2) each theory $\mathcal{T}$ of $\mathcal{L}$ determines a theory $G(\mathcal{T})$ of $\mathcal{L}_0$ made of theorems of $\mathcal{T}$ which are formulae of $\mathcal{L}_0$.

The functions F and G are monotone and they form a Galois connection, denoted $F \dashv G$:

$$\text{Theory}(\mathcal{L}_0) \underset{G}{\overset{F}{\rightleftarrows}} \text{Theory}(\mathcal{L}) \quad (\bot)$$

- for each theory $\mathcal{T}$ of $\mathcal{L}$ and each theory $\mathcal{T}_0$ of $\mathcal{L}_0$, we have:

$$\mathcal{T}_0 \subseteq G(\mathcal{T}) \iff F(\mathcal{T}_0) \subseteq T.$$

⋆ It follows that

$$\mathcal{T}_0 \subseteq G(F(\mathcal{T}_0)) \text{ and } F(G(\mathcal{T})) \subseteq \mathcal{T}.$$

Absolute vs Relative Hilbert-Post completeness

- (Absolute) H-P completeness (wrt to a logic L) A theory T is H-P complete if:
  - at least one sentence is unprovable from T
  - and every theory containing T
    either is T or is made of all sentences

  i.e., T is maximally consistent

- Relative H-P completeness (wrt to two logics $L_0 \subseteq L$) A theory T is relatively H-P complete wrt L_0 if:
  - at least one sentence is unprovable from T
  - and every theory containing T
    can be generated from T and some sentences in L_0

  i.e., T is maximally consistent "up to L_0"

Canonical forms

Proposition

- *For each propagator* $a^{(1)} : X \to Y$*, either* a *is pure or there is a pure term* $v^{(0)} : X \to P$ *such that*

$$a^{(1)} \equiv [\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v^{(0)}.$$

- *For each catcher* $f^{(2)} : X \to Y$*, either* f *is a propagator or there is a propagator* $a^{(1)} : P \to Y$ *and a pure term* $u^{(0)} : X \to P$ *such that*

$$f^{(2)} \equiv a^{(1)} \circ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \circ v^{(0)}.$$

Equivalences between propagators

Proposition

Let us assume that $[\,]_Y^{(0)}$ *is a monomorphism with respect to propagators. A strong equation between two accessor terms is (T-)equivalent to an equation between pure terms:*

$$[\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v_1^{(0)} \equiv [\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v_2^{(0)} \iff v_1^{(0)} \equiv v_2^{(0)}.$$

Equivalences between propagators

Proposition

Let us assume that $[\,]_Y^{(0)}$ *is a monomorphism with respect to propagators. A strong equation between two accessor terms is (T-)equivalent to an equation between pure terms:*

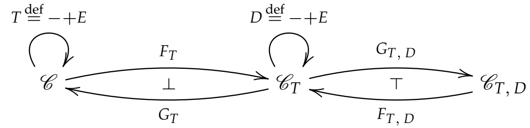
$$[\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v_1^{(0)} \equiv [\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v_2^{(0)} \iff v_1^{(0)} \equiv v_2^{(0)}.$$

Assumption

A strong equation between an accesor and a pure term is "absurd".

$$[\,]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v^{(0)} \equiv v_2^{(0)} \iff (\textit{for all } f^{(0)}, g^{(0)} : X \to Y, f^{(0)} \equiv g^{(0)}).$$

More on absurdity assumption



$\eta : Id \Rightarrow T \qquad F_T \dashv G_T \qquad \varepsilon : T \Rightarrow Id$

$$[\ ]_Y^{(0)} \circ \mathtt{tag}^{(1)} \circ v_1^{(0)} \equiv v_2^{(0)} : X \to Y$$

would be interpreted as

$$\underbrace{T([\ ]_Y) \circ \mu_{\mathbb{0}} \circ T(\mathtt{tag}) \circ T(v_1)}_{f} = \underbrace{T(v_2)}_{g} : X + E \to Y + E.$$

- $\Rightarrow$ $\forall e \in E,\ f(e) = e = g(e)$,
- $\Rightarrow$ $\forall x \in X,\ f(x) = e$ for some $e \in E$ but $g(x) = y$ for some $y \in Y$.

Since "+" is the disjoint union, "=" cannot hold!

absurdity assumption (left-to-right): if f = g holds, then all pure terms collapse!!!

Equivalences between catchers

Proposition

- *A strong equation between catchers is (T-)equivalent to two equations between propagators:*

$$a_1^{(1)} \circ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \circ u_1^{(0)} \equiv a_2^{(1)} \circ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \circ u_2^{(0)}$$

$$\Longleftrightarrow$$

$$a_1^{(1)} \equiv a_2^{(1)} \;\; \textit{and} \;\; a_1^{(1)} \circ u_1^{(0)} \equiv a_2^{(1)} \circ u_2^{(0)}.$$

- *a strong equation between a catcher and an accessor is (T-)equivalent to equations between propagators:*

$$a_1^{(1)} \circ \mathtt{untag}^{(2)} \circ \mathtt{tag}^{(1)} \circ u_1^{(0)} \equiv a_2^{(1)}$$

$$\Longleftrightarrow$$

$$a_1^{(1)} \circ u_1^{(0)} \equiv a_2^{(1)} \;\; \textit{and} \;\; a_1^{(1)} \equiv [\,]_Y^{(0)} \circ \mathtt{tag}^{(1)}.$$

Equivalences between catchers

Theorem

The base theory of exceptions $\mathcal{T}_{exc}$ of the logic $\mathcal{L}_{exc-\oplus}$ is relatively Hilbert-Post complete with respect to the pure logic $\mathcal{L}_{meq+\mathbb{0}}$.

The *relative Hilbert-Post completeness* lifts the *absolute Hilbert-Post completeness* from the logic $\mathcal{L}_0$ to the logic $\mathcal{L}$:

Theorem

$$Theory(\mathcal{L}_0) \underset{G}{\overset{F}{\rightleftarrows}} Theory(\mathcal{L}) \qquad (F \perp G)$$

Let $\mathcal{T}_0$ be a theory of $\mathcal{L}_0$ and $\mathcal{T} = F(\mathcal{T}_0)$.
If

- *$\mathcal{T}_0$ is Hilbert-Post complete (in $\mathcal{L}_0$) and*
- *$\mathcal{T}$ is relatively Hilbert-Post complete with respect to $\mathcal{L}_0$,*

then, $\mathcal{T}$ is Hilbert-Post complete (in $\mathcal{L}$).

The *relative Hilbert-Post completeness* is well suited to the combination of logics:

Lemma

$$\mathit{Theory}(\mathcal{L}_0) \underset{G_1}{\overset{F_1}{\underset{\perp}{\rightleftarrows}}} \mathit{Theory}(\mathcal{L}_1) \underset{G_2}{\overset{F_2}{\underset{\perp}{\rightleftarrows}}} \mathit{Theory}(\mathcal{L}_2)$$

Let $\mathcal{T}_1 = F_1(\mathcal{T}_0)$ *and let* $\mathcal{T}_2 = F_2(\mathcal{T}_1)$.
If

- $\mathcal{T}_1$ *is relatively Hilbert-Post complete with respect to* $\mathcal{L}_0$ *and*
- $\mathcal{T}_2$ *is relatively Hilbert-Post complete with respect to* $\mathcal{L}_1$*,*

then, $\mathcal{T}_2$ *is relatively Hilbert-Post complete with respect to* $\mathcal{L}_0$.