Certification of programs with computational effects

Burak Ekici*
with Jean-Guillaume Dumas*, Dominique Duval* and Damien Pous**
* LJK, University Joseph Fourier, France
** LIP, ENS-Lyon, France

CICM'14
Coimbra, Portugal
July 11, 2014

Motivation

- Verifying properties of programs involving computational (side) effects such as:
  - State
  - Exceptions
  - IO
  - Partiality
  - ...

  through `decorated logic`.

Motivation

- Verifying properties of programs involving computational (side) effects such as:
  - State
  - Exceptions
  - IO
  - Partiality
  - ...

  through `decorated logic`.
- Developing related `Coq libraries` for each effect and composing them.

The State

State of a program:

- the snapshot of the memory locations (variables) at any point during execution
- not `syntactically` mentioned
- can be viewed as `set` or `array` of locations denoted by `S`

x	y	z	t	u	v
1	2	3	4	5	6

- provides an access to itself via an interface:
  - $\mathtt{update}_{\mathtt{x}}(3)$; $\mathtt{lookup}_{\mathtt{x}}$
  - `x = 3; x`

The State

State of a program:

- the snapshot of the memory locations (variables) at any point during execution
- not `syntactically` mentioned
- can be viewed as `set` or `array` of locations denoted by S

x	y	z	t	u	v
1	2	3	4	5	6

- provides an access to itself via an interface:
  - $\mathtt{update}_{\mathtt{x}}(3)$; $\mathtt{lookup}_{\mathtt{x}}$
  - `x = 3; x`

N.B. Any access (for any reason: `update` or `lookup`) to the state is defined as a computational effect.

The mismatch between syntax and interpretation

$\texttt{update}_{\texttt{x}}$:
`x = 3;`

- in syntax: $\texttt{int} \to \texttt{void}$
- in an interpretation: $\texttt{S} \times \texttt{int} \to \texttt{S}$

$\texttt{lookup}_{\texttt{x}}$:
`x`

- in syntax: $\texttt{void} \to \texttt{int}$
- in an interpretation: $\texttt{S} \to \texttt{int}$

How to prove program equivalences

Motivation: Proving program equivalences with no mention of the state!

How to prove program equivalences

Motivation: Proving program equivalences with no mention of the state!

An approach: Decorated logic for states

Decorated proofs for computational effects: States. Dumas et al.'12

How to prove program equivalences

Motivation: Proving program equivalences with no mention of the state!

An approach: Decorated logic for states

Decorated proofs for computational effects: States. Dumas et al.'12

- to have proofs independent of the state
  - decorations in use

Decorated Logic for states

The decorated logic $\mathcal{L}_{sts}$ on a category $\mathbb{C}$ with cartesian products:

Grammar

Types: t	$::=$	$A \mid B \mid \cdots \mid t \times t \mid \mathbb{1} \mid V_T \;_{s.t. T \in Loc}$
Terms: f	$::=$	$id \mid f \circ f \mid \langle f, f \rangle \mid \pi_1 \mid \pi_2 \mid \langle\ \rangle \mid$
		$\texttt{lookup}_T : \mathbb{1} \to V_T \mid \texttt{update}_T : V_T \to \mathbb{1}$
Decoration for terms: (d)	$::=$	$(0) \mid (1) \mid (2)$
Equations: e	$::=$	$f \equiv f \mid f \sim f$

Decorated logic for states: decorations

Let T be a location and V_T be the set of values that can be stored in T.

E.g., $V_T = \texttt{int}$

Terms are classified and decorated:

- pure: e.g., $\texttt{id}_{\texttt{V}_\texttt{T}}^{(0)} : V_T \to V_T$, $\texttt{forget}_{\texttt{T}}^{(0)} : V_T \to \mathbb{1}$
- accessors: e.g., $\texttt{lookup}_{\texttt{T}}^{(1)} : \mathbb{1} \to V_T$
- modifiers: e.g., $\texttt{update}_{\texttt{T}}^{(2)} : V_T \to \mathbb{1}$

Decorated logic for states: decorations

Let T be a location and V_T be the set of values that can be stored in T.

E.g., $V_T = \mathtt{int}$

Terms are classified and decorated:

- pure: e.g., $\mathtt{id}_{\mathtt{V_T}}^{(0)} : V_T \to V_T$, $\mathtt{forget}_{\mathtt{T}}^{(0)} : V_T \to \mathbb{1}$
- accessors: e.g., $\mathtt{lookup}_{\mathtt{T}}^{(1)} : \mathbb{1} \to V_T$
- modifiers: e.g., $\mathtt{update}_{\mathtt{T}}^{(2)} : V_T \to \mathbb{1}$

N.B. `Decorations` specify the effects of the terms on the state.

Decorated logic for states: equations

Equations: $\mathtt{f} = \mathtt{g} : \mathtt{X} \to \mathtt{Y}$

Decorations on equations:

- $\mathtt{f}^{(2)} \equiv \mathtt{g}^{(2)}$ if the equation is strong (result + effect equality)
- $\mathtt{f}^{(2)} \sim \mathtt{g}^{(2)}$ if the equation is weak (result equality)

Rules of decorated logic for states

- Conversion rules

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

- Equivalence rules
- Rules on monadic equational logic
- Categorical product rules

Rules cont'd

- Effect rule

$$(\text{st-effect-u})\,\frac{f^{(2)}, g^{(2)} : A \to B \quad f^{(2)} \sim g^{(2)} \quad \langle\,\rangle_A^{(0)} \circ f^{(2)} \equiv \langle\,\rangle_A^{(0)} \circ g^{(2)}}{f \equiv g}$$

Rules cont'd

- Effect rule

$$(\text{st-effect-u})\,\frac{f^{(2)}, g^{(2)} : A \to B \quad f^{(2)} \sim g^{(2)} \quad \langle\,\rangle_A^{(0)} \circ f^{(2)} \equiv \langle\,\rangle_A^{(0)} \circ g^{(2)}}{f \equiv g}$$

- Axioms (for each $T \in Loc$)

$$\texttt{lookup}_T^{(1)} \circ \texttt{update}_T^{(2)} \sim id_{V_T}^{(0)}$$

Coq formalization of states effect: terms

Coq implementation of terms:

```
Inductive term: Type → Type → Type :=
  | comp : ∀ {X Y Z: Type}, term X Y → term Y Z → term X Z
  | pair : ∀ {X Y Z: Type}, term X Z → term Y Z → term (X×Y) Z
  | tpure : ∀ {X Y: Type}, (X → Y) → term Y X
  | lookup : ∀ i:Loc, term (Val i) unit
  | update : ∀ i:Loc, term unit (Val i)
 Infix "o" := comp (at level 60).
```

- term is a `dependent type`.

Coq formalization: decorations

Decorations:

```
Inductive kind := pure | ro | rw.
```

Coq formalization: decorations

Decorations:

```
Inductive kind := pure | ro | rw.
```

Term decorations:

```
Inductive is: kind → ∀ X Y, term X Y → Prop :=
 | is_tpure: ∀ X Y (f: X → Y), is pure (@tpure X Y f)
 | is_comp: ∀ 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: ∀ k X Y Z (f: term X Z) (g: term Y Z), is k f → is k g → is k (pair f g)
 | is_lookup: ∀ i, is ro (lookup i)
 | is_update: ∀ i, is rw (update i)
 | is_pure_ro: ∀ X Y (f: term X Y), is pure f → is ro f
 | is_ro_rw: ∀ X Y (f: term X Y), is ro f → is rw f.
Hint Constructors is.
```

Coq formalization: rules

Rules w.r.t. strong equality:

```
Reserved Notation "x == y" (at level 80).

Inductive strong: ∀ X Y, relation (term X Y) :=
                    ⋮
      | effect_rule: ∀ X Y (f g: term Y X), forget o f == forget o g → f ∼ g → f == g
                    ⋮
```

Coq formalization: rules cont'd

Rules w.r.t. weak equality:

```
Reserved Notation "x ∼ y" (at level 80).

with weak: ∀ X Y, relation (term X Y) :=
                        ⋮
         | axiom_1: ∀ i, lookup i o update i ∼ id
                        ⋮
```

The generic states library

Proofs of 7 properties of the `state` through propositions by Plotkin et al'02.

E.g.,

- **`Commutation update-update:`**
  $\forall i \neq j \in Loc,\ u_j \circ \pi_2 \circ (u_i \rtimes id_j) \equiv u_i \circ \pi_1 \circ (id_i \ltimes u_j) : V_i \times V_j \to \mathbb{1}$

source code (STATES)

The IMP-STATES library

Soundness property check for STATES: a new version, IMP-STATES.

IMP Syntax

$$\begin{array}{lll} \text{aexp: } a_1 \ a_2 & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \\ \text{bexp: } b_1 \ b_2 & ::= & tt \mid ff \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 > a_2 \mid a_1 < a_2 \mid \\ & & b_1 \wedge b_2 \mid b_1 \vee b_2 \\ \text{cmd: } c_1 \ c_2 & ::= & skip \mid x := e \mid c_1 ; c_2 \mid if \ b \ then \ c_1 \ else \ c_2 \mid while \ b \ do \ c_1 \end{array}$$

Operational semantics of IMP language is defined through `decorated logic`.

Programs can be proven

IMP programs with sequences, assignments, conditionals and terminating loops:

E.g.,

```
prog_1 = (
 if b then (
   if b then (f)
   else (g)
  )
 else (h)
) .
```

==

```
prog_2 = (
 if b then (f)
 else (h)
) .
```

```
prog_3 = (
  var x ;
  x := 2 ;
  while (x < 11) do (
    x := x + 4 ;
  )
) .
```

==

```
prog_4 = (
  var x ;
  x := 14 ;
) .
```

source code (IMP-STATES)

The decorated logic for exceptions

The decorated logic $\mathcal{L}_{exc}$ on a category $\mathbb{C}$ with disjoint union:

Grammar

Types: t	$::=$	$A \mid B \mid \cdots \mid t + t \mid \mathbb{0} \mid EV_T \ {}_{s.t. T \in Exn}$
Terms: f	$::=$	$id \mid f \circ f \mid [f \vert f] \mid inl \mid inr \mid [\,] \mid$
		$\mathtt{tag}_T : EV_T \to \mathbb{0} \mid \mathtt{untag}_T : \mathbb{0} \to EV_T$
Decoration for terms: (d)	$::=$	$(0) \mid (1) \mid (2)$
Equations: e	$::=$	$f \equiv f \mid f \sim f$

$\mathcal{L}_{exc}$ is dual to $\mathcal{L}_{sts}$.

- generic Coq library to cope with `exceptions effect`.

A duality between exceptions and states. Dumas et al'12

source code (EXCEPTIONS)

The decorated logic for states + exceptions

The combined decorated logic, $\mathcal{L}_{sts\oplus exc}$, on a distributive category $\mathbb{C}$:

Grammar

Types: t	$::=$	$merged$
Terms: f	$::=$	$merged$
Decoration for terms: (d^s, d^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: e	$::=$	$f \equiv\equiv f \mid f \equiv\sim f \mid f \sim\equiv f \mid f \sim\sim f$

Rules are merged. $\Longrightarrow$ A `generic Coq library`.

source code (STATES-EXCEPTIONS)

The IMP-STATES-EXCEPTIONS library

An extension to IMP-STATES: IMP-STATES-EXCEPTIONS.

IMP Syntax

$$\begin{array}{lcl}
\text{aexp: } a_1\ a_2 & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \\
\text{bexp: } b_1\ b_2 & ::= & tt \mid ff \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 > a_2 \mid a_1 < a_2 \mid \\
 & & b_1 \wedge b_2 \mid b_1 \vee b_2 \\
\text{cmd: } c_1\ c_2 & ::= & skip \mid x := e \mid c_1 ; c_2 \mid if\ b\ then\ c_1\ else\ c_2 \mid while\ b\ do\ c_1 \mid \\
 & & \color{red}{throw}\ exc \mid \color{red}{try}\ c_1\ \color{red}{catch}\ exc \Rightarrow c_2
\end{array}$$

source code (IMP-STATES-EXCEPTIONS)

Programs can be proven

Programs with additional structures: `throw` and `try-catch` blocks.

E.g.,

```
prog_5 = (
  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) ;
  y := 15 ;
) .
```

==

```
prog_6 = (
 var x, y ;
 x := 0 ; y := 15 ;
) .
```

Program calculating the rank of a (2×2) matrix modulo composite numbers:

```
prog_7= (
  var a, b, c, d, m ;
  var r ;
  var  t, u, u1, q, g, g1 ;
  a := 2; b := 1 ; c := 3 ; d := 4 ; m := 6 ;
  if(a = 0) then(
      t := a ; a := b ; b := t ;
      t := c ; c := d ; d := t ;
    )
  else skip ;
  if(a = 0) then(
      t := a ; a := c ; c := t ;
      t := b ; b := d ; d := t ;
    )
  else skip ;
  if(a = 0) then(
      if(b = 0) then r := 0 ;
      else r := 1 ;
    )
  else(
    try(
      u := 0 ; u1 := 1 ; g1 := a; g := m ;
      while(g1 > 0) do(
        q := g / g1 ;
        t := u - q * u1 ; u := u1 ; u1 := t ;
        t := g - q * g1 ; g := g1 ; g1 := t ;
      )
    )
      if not (g = 1) then throw e ;
      else skip ;
    catch e => (
      m := m / g ;
      u := 0 ; u1 := 1 ; g1 := a ; g := m ;
      while(g1 > 0) do(
        q := g / g1 ;
        t := u - q * u1 ; u := u1 ; u1 := t ;
        t := g - q * g1 ; g := g1 ; g1 := t ;
      )
    )
      d := (d - u * c * b) % m ;
      if(d = 0) then r := 1 ;
      else r := 2 ;
  )
) .
```

==

```
prog_8= (
  var a, b, c, d, m ;
  var r ;
  var  t, u, u1, q, g, g1 ;
  a := 2 ; u1 := 3 ; q := 2 ; g := 1 ;
  t := 0 ; g1 := 0 ; c := 3 ;  u := -1 ;
  b := 1 ; m := 3 ; d := 1 ;  r := 2 ;
).
```

Program calculating the rank of a (2×2) matrix modulo composite numbers:

```
prog_7= (
  var a, b, c, d, m ;
  var r ;
  var  t, u, u1, q, g, g1 ;
  a := 2; b := 1 ; c := 3 ; d := 4 ; m := 6 ;
  if(a = 0) then(
      t := a ; a := b ; b := t ;
      t := c ; c := d ; d := t ;
    )
  else skip ;
  if(a = 0) then(
      t := a ; a := c ; c := t ;
      t := b ; b := d ; d := t ;
    )
  else skip ;
  if(a = 0) then(
      if(b = 0) then r := 0 ;
      else r := 1 ;
    )
  else(
    try(
      u := 0 ; u1 := 1 ; g1 := a; g := m ;
      while(g1 > 0) do(
        q := g / g1 ;
        t := u - q * u1 ; u := u1 ; u1 := t ;
        t := g - q * g1 ; g := g1 ; g1 := t ;
      )
    )
      if not (g = 1) then throw e ;
      else skip ;
    catch e => (
      m := m / g ;
      u := 0 ; u1 := 1 ; g1 := a ; g := m ;
      while(g1 > 0) do(
        q := g / g1 ;
        t := u - q * u1 ; u := u1 ; u1 := t ;
        t := g - q * g1 ; g := g1 ; g1 := t ;
      )
    )
      d := (d - u * c * b) % m ;
      if(d = 0) then r := 1 ;
      else r := 2 ;
    )
  ) .
```

==

```
prog_8= (
  var a, b, c, d, m ;
  var r ;
  var  t, u, u1, q, g, g1 ;
  a := 2 ; u1 := 3 ; q := 2 ; g := 1 ;
  t := 0 ; g1 := 0 ; c := 3 ;  u := -1 ;
  b := 1 ; m := 3 ; d := 1 ;  r := 2 ;
).
```

Consider:

- '/' is the `integer division`
- '%' is the `modulo reduction`

See J.-G. Dumas' CICM'14 slides.

So far

- A Coq library for the global states:
  - with Hilbert-Post Completeness proof
- A Coq library for exceptions
- A Coq library for combined states and exceptions
- IMP specifications:
  - IMP-STATES
  - IMP-STATES-EXCEPTIONS
- All sources on `http://coqeffects.forge.imag.fr/`

What's next?

- Improving the way: effect combination
  - developing an "tool" in Coq to combine separate effects.
- Interpreting `Hoare Logic` in `decorated settings`
- Having modularity: interpreting functions

The end!

`Many thanks for your kind attention!`

`Questions?`

Accessors interpreted

Below stated procedure is used to interpret accessors in decorated settings:

Procedure 1: accessors: interpretation of decorated accessors.

Data: The category $\mathbb{C}$ with a distinguished "object of states $\mathtt{S}$", etc...

Result: the interpretations of accessors via states comonad.

Prove $\Phi \colon \mathbb{C} \to \mathbb{C}$ as an endo-functor

$$\Phi(\mathtt{X}) = \mathtt{X} \times \mathtt{S}$$

$$\Phi(\mathtt{f} \colon \mathtt{X} \to \mathtt{Y}) = (\mathtt{f} \times \mathtt{id}_{\mathtt{S}}) \colon \mathtt{X} \times \mathtt{S} \to \mathtt{Y} \times \mathtt{S}$$

Prove cM $(\Phi, \delta \colon \Phi \Rightarrow \Phi^2, \epsilon \colon \Phi \Rightarrow \mathtt{id}_{\mathbb{C}})$ as the states comonad.

Construct the coKleisli category $\mathbb{C}_1$ of cM over $\mathbb{C}$

$$\mathtt{Obj}(\mathbb{C}_1) = \mathtt{Obj}(\mathbb{C})$$

$$\mathtt{Hom}_{(\mathbb{C}_1)}(\mathtt{X}, \mathtt{Y}) = \mathtt{Hom}_{(\mathbb{C})}(\Phi\mathtt{X}, \mathtt{Y})$$

$$\forall \mathtt{f}^* : \mathtt{X} \to \mathtt{Y}\ \mathtt{g}^* : \mathtt{Y} \to \mathtt{Z},\ \mathtt{g}^* \circ_{(\mathbb{C}_1)} \mathtt{f}^* = (\mathtt{g} \circ_{(\mathbb{C})} \Phi\mathtt{f} \circ_{(\mathbb{C})} \delta) \in \mathbb{C}.$$

return *Any impure morphism* $\mathtt{f}^{(1)} : \mathtt{X} \to \mathtt{Y} \in \mathtt{Hom}_{(\mathbb{C}_1)}$ *is interpreted as* $\mathtt{f}_0 \colon \mathtt{X} \times \mathtt{S} \to \mathtt{Y} \in \mathtt{Hom}_{(\mathbb{C})}$ *which represents an accessor.*

Modifiers interpreted

Interpretation of decorated modifiers goes even beyond the one for accessors:

Procedure 2: `modifiers`: interpretations of decorated modifiers.

Data: Categories $\mathbb{C}$ and $\mathbb{C}_1$ as before.
Result: the interpretations of modifiers via monad $\mathtt{M}$.

Prove $\Phi_1 : \mathbb{C}_1 \to \mathbb{C}_1$ as an endo-functor

$$\Phi_1(\mathtt{X}) = \mathtt{X} \times \mathtt{S}$$

$$\Phi_1(\mathtt{f}^* : \mathtt{X} \to \mathtt{Y}) = (\mathtt{f}^* \times \mathtt{id}_\mathtt{S}) : \mathtt{X} \times \mathtt{S} \to \mathtt{Y} \times \mathtt{S}$$

Prove $\mathtt{M}$ $(\Phi_1, \mu \colon \Phi_1^2 \Rightarrow \Phi_1, \eta \colon \mathtt{id}_{\mathbb{C}_1} \Rightarrow \Phi_1)$ as a monad.

Construct the `Kleisli` category $\mathbb{C}_2$ of $\mathtt{M}$ over $\mathbb{C}_1$

$$\mathtt{Obj}(\mathbb{C}_2) = \mathtt{Obj}(\mathbb{C}_1) = \mathtt{Obj}(\mathbb{C})$$

$$\mathtt{Hom}_{(\mathbb{C}_2)}(\mathtt{X}, \mathtt{Y}) = \mathtt{Hom}_{(\mathbb{C}_1)}(\mathtt{X}, \Phi_1 \mathtt{Y}) = \mathtt{Hom}_{(\mathbb{C})}(\Phi \mathtt{X}, \Phi \mathtt{Y})$$

$$\forall \mathtt{f}^+ : \mathtt{X} \to \mathtt{Y}\ \mathtt{g}^+ : \mathtt{Y} \to \mathtt{Z},\ \mathtt{g}^+ \circ_{(\mathbb{C}_2)} \mathtt{f}^+ = (\mu_\mathtt{Y} \circ_{(\mathbb{C}_1)} \Phi_1 \mathtt{g}^* \circ_{(\mathbb{C}_1)} \mathtt{f}^*) \in \mathbb{C}_1.$$

return *Any impure morphism* $\mathtt{f}^{(2)} : \mathtt{X} \to \mathtt{Y} \in \mathtt{Hom}_{(\mathbb{C}_2)}$ *is interpreted as* $\mathtt{f}_0 : \mathtt{X} \times \mathtt{S} \to \mathtt{Y} \times \mathtt{S} \in \mathtt{Hom}_{(\mathbb{C})}$ *which represents a modifier.*

Derived terms for states

Some derived pure terms:

```
Definition id {X: Type} : term X X := tpure id.
Definition pi1 {X Y: Type} : term X (X×Y) := tpure fst.
Definition pi2 {X Y: Type} : term Y (X×Y) := tpure snd.
Definition forget {X} : term unit X := tpure (fun _ ⇒ tt).
Definition constant {X: Type} (v: X): term X unit := tpure (fun _ ⇒ v).
Definition permut {X Y}: term (X×Y) (Y×X) := pair pi2 pi1.
Definition loopdec (b: term (unit+unit) unit) (f: term unit unit) : term unit unit := tpure  (fun tt ⇒ tt).
```

Some derived pairs:

```
Definition perm_pair {X Y Z} (f: term Y X) (g: term Z X): term (Y×Z) X
:= permut o pair g f.
Definition prod {X Y X' Y'} (f: term X X') (g: term Y Y'): term (X×Y) (X'×Y')
:= pair (f o pi1) (g o pi2).
Definition perm_prod {X Y X' Y'} (f: term X X') (g: term Y Y')
:= permut o pair (g o pi2) (f o pi1).
```

IMP semantics over decorated logic

Arithmetic and boolean expression declaration:

```
Inductive Exp : Type → Type :=
 | const : ∀ A, A → Exp A
 | loc : Loc → Exp Z
 | apply : ∀ A B, (A → B) → Exp A → Exp B
 | pairExp : ∀ A B, Exp A → Exp B → Exp (A × B).
```

```
Fixpoint defExp A (e: Exp A): term A unit :=
 match e with
   | const Z n ⇒ constant n
   | loc x ⇒ lookup x
   | apply _ _ f x ⇒ tpure f o (defExp x)
   | pairExp _ _ x y ⇒ pair (defExp x) (defExp y)
 end.
```

IMP semantics over decorated logic

Arithmetic and boolean expression declaration:

```
Inductive Exp : Type → Type :=
 | const : ∀ A, A → Exp A
 | loc : Loc → Exp Z
 | apply : ∀ A B, (A → B) → Exp A → Exp B
 | pairExp : ∀ A B, Exp A → Exp B → Exp (A × B).
```

```
Fixpoint defExp A (e: Exp A): term A unit :=
 match e with
   | const Z n ⇒ constant n
   | loc x ⇒ lookup x
   | apply _ _ f x ⇒ tpure f o (defExp x)
   | pairExp _ _ x y ⇒ pair (defExp x) (defExp y)
 end.
```

A Coq side pure function: add

```
Definition add (p: Z × Z): Z :=
  match p with
    | (x, y) => x + y
  end.
```

Pure Coq functions in operation

```
Check apply add (pairExp (const 30) (const 40)): Exp Z.

Check defExp(apply add (pairExp (const 30) (const 40))) =
      tpure add o (pair (constant 30) (constant 40)) : term Z unit.

| IMP_IntAop: ∀ (p q: Z) (f: Z×Z → Z),
            tpure f o (pair (@constant Z p) (@constant Z q)) == (constant (f(p,q)))
```

IMP semantics over decorated logic cont'd

Command declaration

```
Inductive Command : Type :=
  | skip : Command
  | sequence : Command → Command → Command
  | assign : Loc → Exp Z → Command
  | ifthenelse : Exp bool → Command → Command → Command
  | loops : Exp bool → Command → Command
```

IMP semantics over decorated logic cont'd

Command declaration

```
Inductive Command : Type :=
    | skip : Command
    | sequence : Command → Command → Command
    | assign : Loc → Exp Z → Command
    | ifthenelse : Exp bool → Command → Command → Command
    | loops : Exp bool → Command → Command
```

Command typed instance definition

```
Fixpoint defCommand (c: Command): (term unit unit) :=
 match c with
    | skip ⇒ (@id unit)
    | sequence c0 c1 ⇒ (defCommand c1) o (defCommand c0)
    | assign j e0 ⇒ (update j) o (defExp e0)
    | ifthenelse b c2 c3 ⇒ copair (defCommand c2) (defCommand c3)
                                          o (passbool o (defExp b))
    | loops b c4 ⇒ (copair (loopdec (passbool o (defExp b)) (defCommand c4)
                                          o (defCommand c4)) (@id unit)) o (passbool o (defExp b))
```

Derived terms for exceptions

Derived terms for exceptions

```
Definition coproj1 {X Y} : term (X+Y) X := tpure inl.
Definition coproj2 {X Y} : term (X+Y) Y := tpure inr.
Definition empty {X} (x: X) : term X Empty_set := tpure (fun _ => x).

Definition ttrue : term (unit+unit) unit := coproj1.
Definition ffalse : term (unit+unit) unit := coproj2.

Definition throw {T1} (t1: T1) (T2: EName): term T1 unit := (empty t1) o tag T2.

Definition eiso {X} : term (X+Empty_set) X := (@coproj1 X Empty_set).
Definition eiso_2 {X} : term (Empty_set+X) X := (@coproj2 Empty_set X).
Definition epermut {X Y} : term (X+Y) (Y+X) := copair coproj2 coproj1.

Definition TRY_CATCH (X Y: Type) (t: EName) (f: term Y X) (g: term Y unit)
  := downcast(copair (@id Y) (g o untag t) o eiso o f).

Definition TRY_CATCH_PARAM_2 (X Y: Type) (t s : EName) (f: term Y X)
                              (g: term Y unit) (h: term Y unit)
  := downcast((copair (@id Y) ((copair g (h o untag s)) o eiso o untag t)) o eiso o f).
```

Comparison in decorated settings

```
{{x := n ;
  if (x >= m ) ...}}
```

$$
\begin{tikzcd}
 & & & \mathbb{Z} \\
\mathbb{1} \arrow[r, "const_n"] & \mathbb{Z} \arrow[r, "update_x"] & \mathbb{1} \arrow[ur, "lookup_x"] \arrow[r, dashed, "<\_,\_>"] \arrow[dr, "const_m"'] & \mathbb{Z}^2 \arrow[u, "\pi_1"'] \arrow[d, "\pi_2"] \\
 & & & \mathbb{Z}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & & \mathbb{Z} \\
\mathbb{1} \arrow[r, "const_n"] & \mathbb{Z} \arrow[r, "update_x"] & \mathbb{1} \arrow[ur, "const_n"] \arrow[r, dashed, "<\_,\_>"] \arrow[dr, "const_m"'] & \mathbb{Z}^2 \arrow[u, "\pi_1"'] \arrow[d, "\pi_2"] \\
 & & & \mathbb{Z}
\end{tikzcd}
$$

Left hand side diagram is `strongly equal` to the right hand side one.

Conditionals in decorated settings

```
{{x := n ;
  if ( x >= m) then f else g }}
```

$$\begin{tikzcd}
 & \mathbb{1} \arrow[d, "b:ttrue"'] \arrow[dr, "f"] & \\
\mathbb{Z}^2 \arrow[r, ">="] & \mathbb{1}+\mathbb{1} \arrow[r, "{[f|g]}"] & \mathbb{1} \\
 & \mathbb{1} \arrow[u, "b:ffalse"] \arrow[ur, "g"'] &
\end{tikzcd}$$

If `b` is `ttrue` then `f` else `g` is executed.

Loops in decorated settings

```
{{x := n ;
  while ( x >= m) do f }}
```

$$
\begin{tikzcd}
 & \mathbb{1} \arrow[d, "b{:}ttrue"'] \arrow[r, "f"] & \mathbb{1} \arrow[dr, "\mathrm{loopdec\ b\ f}"] & \\
\mathbb{Z}^2 \arrow[r, ">="] & \mathbb{1}+\mathbb{1} \arrow[rr, "{[\mathrm{loopdec\ b\ f} \circ f | id_{\mathbb{1}}]}"] & & \mathbb{1} \\
 & \mathbb{1} \arrow[u, "b{:}ffalse"] \arrow[urr, "id_{\mathbb{1}}"'] & &
\end{tikzcd}
$$

`Replace` loopdec with the whole diagram as long as `b` is `ttrue`. `Quit` the loop through $id_{\mathbb{1}}$ when it becomes `ffalse`.

throw & try-catch in decorated settings

`{{throw e}}`.

$$\mathbb{1} \xrightarrow{tag_e} \emptyset \xrightarrow{[\,]_{\mathbb{1}}} \mathbb{1}$$

throwing an exception of name e.

`{{try f catch e ⇒ g}}`.

$$\begin{tikzcd}
& \mathbb{1} \arrow[d, "id_{\mathbb{1}}"'] \arrow[drr, "id_{\mathbb{1}}"] & & \\
\mathbb{1} \arrow[r, "f"] & \mathbb{1} + \emptyset \arrow[rr, "{[id|k]}"] & & \mathbb{1} \\
& \emptyset \arrow[u, "id_{\emptyset}"] \arrow[urr, "k"'] & &
\end{tikzcd}
\qquad
\emptyset \xrightarrow{untag_e} \mathbb{1} \xrightarrow{g} \mathbb{1}$$

try f:

1. if everything is ordinary, then catch block is not executed.
2. if it throws an exception of name e then it is caught (recovered) inside the catch block and g is executed.
3. if it throws an exception of name $e^* \neq e$ then, e^* gets propagated (by g).

Hoare logic

Formal system to reason about programs with `states` `effect`:

$$(\textsc{Skip})\{\phi\}\,\texttt{skip}\,\{\phi\} \qquad (\textsc{Assign})\{\phi[a/x]\}\,x := a\,\{\phi\}$$

$$(\textsc{Seq})\,\frac{\{\phi\}\,c_1\,\{\psi\} \quad \{\psi\}\,c_2\,\{\sigma\}}{\{\phi\}\,c_1; c_2\,\{\sigma\}} \qquad (\textsc{Cond})\,\frac{\{b \wedge \phi\}\,c\,\{\psi\} \quad \{\neg b \wedge \phi\}\,d\,\{\psi\}}{\{\phi\}\,\texttt{if b then c else d}\,\{\psi\}}$$

$$(\textsc{Loop})\,\frac{\{b \wedge \phi\}\,c\,\{\phi\}}{\{\phi\}\,\texttt{while b do c}\,\{\phi \wedge \neg b\}} \qquad (\textsc{Conseq})\,\frac{\phi \to \phi' \quad \{\phi'\}\,c\,\{\psi'\} \quad \psi' \to \psi}{\{\phi\}\,c\,\{\psi\}}$$

Hoare logic for `exceptions`: See ▸ lecture notes by Claude Marché