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.
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 $S$

$\begin{array}{cccccc}
 x & y & z & t & u & v \\
 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}$

- provides an access to itself via an interface:
  - $update_x(3)$; $lookup_x$
  - $x = 3$; $x$
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 $S$

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- provides an access to itself via an interface:
  - $update_x(3)$; $lookup_x$
  - $x = 3$; $x$

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

\[ \text{update}_x: \]
\[ x = 3; \]
\[ \bullet \text{ in syntax: } \texttt{int} \rightarrow \texttt{void} \]
\[ \bullet \text{ in an interpretation: } S \times \texttt{int} \rightarrow S \]

\[ \text{lookup}_x: \]
\[ x \]
\[ \bullet \text{ in syntax: } \texttt{void} \rightarrow \texttt{int} \]
\[ \bullet \text{ in an interpretation: } S \rightarrow \texttt{int} \]
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
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{I} \mid V_T \text{ s.t. } T \in \text{Loc}$

**Terms:** $f$ ::=$ \text{id} \mid f \circ f \mid \langle f, f \rangle \mid \pi_1 \mid \pi_2 \mid \langle \rangle \mid$

$\text{lookup}_T : \mathbb{I} \rightarrow V_T \mid \text{update}_T : V_T \rightarrow \mathbb{I}$

**Decoration for terms:** $(d)$ ::=$ (0) \mid (1) \mid (2)$

**Equations:** $e$ ::=$ f \equiv f \mid f \sim f$
Let $T$ be a location and $V_T$ be the set of values that can be stored in $T$.

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

Terms are classified and decorated:

- **pure**: e.g., $\text{id}_{V_T}^{(0)} : V_T \to V_T$, $\text{forget}_{T}^{(0)} : V_T \to \bot$

- **accessors**: e.g., $\text{lookup}_{T}^{(1)} : \bot \to V_T$

- **modifiers**: e.g., $\text{update}_{T}^{(2)} : V_T \to \bot$
Let $T$ be a location and $V_T$ be the set of values that can be stored in $T$.

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

Terms are classified and decorated:

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

N.B. Decorations specify the **effects** of the terms on the state.
Decorated logic for states: equations

Equations: \( f = g : X \to Y \)

Decorations on equations:

- \( f^{(2)} \equiv g^{(2)} \) if the equation is strong (result + effect equality)
- \( f^{(2)} \sim g^{(2)} \) if the equation is weak (result equality)
Rules of decorated logic for states

- **Conversion rules**

\[
\begin{align*}
\frac{f(0)}{f(1)} & \quad \frac{f(1)}{f(2)} & \quad \frac{f(d) \equiv g(d')}{f \sim g} & \quad \frac{f(d') \sim g(d')}{f \equiv g} \quad \text{if } d' \leq 1
\end{align*}
\]

- **Equivalence rules**
- **Rules on monadic equational logic**
- **Categorical product rules**
Rules cont’d

- Effect rule

\[
\frac{f^{(2)}, g^{(2)}: A \rightarrow B \quad f^{(2)} \sim g^{(2)} \quad \langle \rangle_{A}^{(0)} \circ f^{(2)} \equiv \langle \rangle_{A}^{(0)} \circ g^{(2)}}{f \equiv g}
\]
• Effect rule

\[
\begin{align*}
\text{Effect rule} & \quad (st\text{-}effect\text{-}u) \quad f^{(2)}, g^{(2)} : A \to B, \quad f^{(2)} \sim g^{(2)}, \quad \langle \rangle^{(0)}_A \circ f^{(2)} \equiv \langle \rangle^{(0)}_A \circ g^{(2)} \\
& \quad \implies f \equiv g
\end{align*}
\]

• Axioms (for each \( T \in \text{Loc} \))

\[
\text{lookup}^{(1)}_T \circ \text{update}^{(2)}_T \sim \text{id}^{(0)}_{V_T}
\]
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: \( \forall X Y, \text{relation (term } X Y) =: \)

\[ \vdots \]

| effect_rule: \( \forall X Y (f g: \text{term } Y X), \text{forget o f == forget o g} \rightarrow f \sim g \rightarrow f == g \)

\[ \vdots \]
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
   ...
   ...

...
Proofs of 7 properties of the state through propositions by Plotkin et al’02.
E.g.,

- Commutation update-update:
  \[
  \forall i \neq j \in \text{Loc}, \ u_j \circ \pi_2 \circ (u_i \times \text{id}_j) \equiv u_i \circ \pi_1 \circ (\text{id}_i \times u_j) : V_i \times V_j \to \mathbb{I}
  \]

(source code)
The IMP-STATES library

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

**IMP Syntax**

\[
\begin{align*}
\text{aexp: } & a_1 \ a_2 \ ::= \ n \ | \ x \ | \ a_1 + a_2 \ | \ a_1 \times a_2 \\
\text{bexp: } & b_1 \ b_2 \ ::= \ tt \ | \ ff \ | \ a_1 = a_2 \ | \ a_1 \neq a_2 \ | \ a_1 > a_2 \ | \ a_1 < a_2 \\
& \quad \ | \ b_1 \land b_2 \ | \ b_1 \lor b_2 \\
\text{cmd: } & c_1 \ c_2 \ ::= \ \text{skip} \ | \ x := e \ | \ c_1; c_2 \ | \ \text{if } b \ \text{then } c_1 \ \text{else } c_2 \ | \ \text{while } b \ \text{do } c_1
\end{align*}
\]

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.,

```plaintext
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: $t$ :::= $A | B | \cdots | t + t | 0 | EV_T \quad s.t. T \in \text{Exn}$

Terms: $f$ :::= $id | f \circ f | [f|f] | \text{inl} | \text{inr} | [] |$

$\text{tag}_T: EV_T \rightarrow 0 \mid \text{untag}_T: 0 \rightarrow EV_T$

Decoration for terms: $(d)$ :::= $(0) | (1) | (2)$

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*
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) | (0^s, 1^e) | (0^s, 2^e) | (1^s, 0^e) | (1^s, 1^e) |

\((1^s, 2^e) | (2^s, 0^e) | (2^s, 1^e) | (2^s, 2^e)\)

Equations: \( e \) ::= \( f \equiv f \ | \ f \equiv f \ | \ f \sim f \ | \ f \sim f \)

Rules are merged. \( \implies \) A generic Coq library.

(source code) (STATES-EXCEPTIONS)
The IMP-STATES-EXCEPTIONS library

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

**IMP Syntax**

aexp: $a_1 \ a_2 \ ::= \ n \mid x \mid a_1 + a_2 \mid a_1 \times a_2$

bexp: $b_1 \ b_2 \ ::= \ tt \mid ff \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 > a_2 \mid a_1 < a_2 \mid$

$b_1 \land b_2 \mid b_1 \lor b_2$

cmd: $c_1 \ c_2 \ ::= \ skip \mid x := e \mid c_1\ c_2 \mid if\ b\ then\ c_1\ else\ c_2 \mid while\ b\ do\ c_1 \mid$

$throw\ exc \mid try\ c_1\ catch\ exc \Rightarrow c_2$

(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 \((2 \times 2)\) matrix modulo composite numbers:

\[
\text{prog}_7 = \{
\begin{align*}
\text{var } & a, b, c, d, m; \\
\text{var } & r; \\
\text{var } & t, u, u1, q, g, g1; \\
a := 2; & b := 1; c := 3; d := 4; m := 6; \\
\text{if}(a = 0) & \text{ then (} \\
& t := a; a := b; b := t; \\
& t := c; c := d; d := t; \\
\text{)} \text{ else skip; } \\
\text{if}(a = 0) & \text{ then (} \\
& t := a; a := c; c := t; \\
& t := b; b := d; d := t; \\
\text{)} \text{ else skip; } \\
\text{if}(a = 0) & \text{ then (} \\
& \text{if}(b = 0) \text{ then } r := 0; \\
& \text{else } r := 1; \\
\text{)} \text{ else (} \\
& \text{try (} \\
& u := 0; u1 := 1; g1 := a; g := m; \\
\text{)} \text{).} \\
\end{align*}
\]

\[
\text{while}(g1 > 0) \text{ do (} \\
& q := g / g1; \\
& t := u - q \times u1; u := u1; u1 := t; \\
& t := g - q \times g1; g := g1; g1 := t; \\
\text{)} \\
\text{if not (} g = 1 \text{) then throw e; } \\
catch e \Rightarrow ( \\
& m := m / g; \\
& u := 0; u1 := 1; g1 := a; g := m; \\
\text{while}(g1 > 0) \text{ do (} \\
& q := g / g1; \\
& t := u - q \times u1; u := u1; u1 := t; \\
& t := g - q \times g1; g := g1; g1 := t; \\
\text{)} \text{) } \\
\text{) } \\
& d := (d - u \times c \times b) \% m; \\
& \text{if}(d = 0) \text{ then } r := 1; \\
& \text{else } r := 2; \\
\text{).}
\]

\[
\text{prog}_8 = \{
\begin{align*}
\text{var } & a, b, c, d, m; \\
\text{var } & r; \\
\text{var } & t, u, u1, q, g, g1; \\
a := 2; & u1 := 3; q := 2; g := 1; \\
t := 0; & g1 := 0; c := 3; u := -1; \\
b := 1; & m := 3; d := 1; r := 2; \\
\end{align*}
\]
Program calculating the rank of a \((2 \times 2)\) matrix modulo composite numbers:

\[
\text{prog}_7= (\\quad \text{var} \ a, b, c, d, m; \\
\quad \text{var} \ r; \\
\quad \text{var} \ t, u, u1, q, g, g1; \\
\quad a := 2; b := 1; c := 3; d := 4; m := 6; \\
\quad \text{if}(a = 0) \text{ then}( \\
\qquad t := a; a := b; b := t; \\
\qquad t := c; c := d; d := t; \\
\qquad ) \\
\quad \text{else skip; } \\
\quad \text{if}(a = 0) \text{ then}( \\
\qquad t := a; a := c; c := t; \\
\qquad t := b; b := d; d := t; \\
\qquad ) \\
\quad \text{else skip; } \\
\quad \text{if}(a = 0) \text{ then}( \\
\qquad \text{if}(b = 0) \text{ then } r := 0; \\
\qquad \text{else } r := 1; \\
\qquad ) \\
\quad \text{else( } \\
\text{ try(} \\
\qquad u := 0; u1 := 1; g1 := a; g := m; \\
\text{ ) } \\
\text{ )}. \\
\]

\[
\text{while}(g1 > 0) \text{ do(} \\
\qquad q := g / g1; \\
\qquad t := u - q * u1; u := u1; u1 := t; \\
\qquad t := g - q * g1; g := g1; g1 := t; \\
\text{ ) } \\
\text{if not } (g = 1) \text{ then throw } e; \\
\text{else skip; } \\
\text{catch } e \Rightarrow ( \\
\qquad m := m / g; \\
\qquad u := 0; u1 := 1; g1 := a; g := m; \\
\text{while}(g1 > 0) \text{ do(} \\
\qquad q := g / g1; \\
\qquad t := u - q * u1; u := u1; u1 := t; \\
\qquad t := g - q * g1; g := g1; g1 := t; \\
\text{ ) } \\
\text{ ) } \\
\quad d := (d - u * c * b) \% m; \\
\text{if}(d = 0) \text{ then } r := 1; \\
\text{else } r := 2; \\
\text{)}. \\
\]

\[
\text{prog}_8= (\\quad \text{var} \ a, b, c, d, m; \\
\quad \text{var} \ r; \\
\quad \text{var} \ t, u, u1, q, g, g1; \\
\quad a := 2; u1 := 3; q := 2; g := 1; \\
\quad t := 0; g1 := 0; c := 3; u := -1; \\
\quad b := 1; m := 3; d := 1; r := 2; \\
\text{)}. \\
\]

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
Many thanks for your kind attention!

Questions?
Appendix

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 $S$”, etc...

**Result:** the interpretations of accessors via states comonad.

1. Prove $\Phi: \mathbb{C} \to \mathbb{C}$ as an endo-functor
   
   $\Phi(X) = X \times S$

   $\Phi(f: X \to Y) = (f \times \text{id}_S): X \times S \to Y \times S$

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

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

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

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

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

5. **return** Any impure morphism $f^{(1)}: X \to Y \in \text{Hom}_{\mathbb{C}_1}$ is interpreted as $f_0: X \times S \to Y \in \text{Hom}_{\mathbb{C}}$ 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 \( \mathbb{M} \).

1. Prove \( \Phi_1 : \mathbb{C}_1 \to \mathbb{C}_1 \) as an endo-functor
   \[ \Phi_1(X) = X \times S \]
   \[ \Phi_1(f^* : X \to Y) = (f^* \times \text{id}_S) : X \times S \to Y \times S \]

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

3. Construct the Kleisli category \( \mathbb{C}_2 \) of \( \mathbb{M} \) over \( \mathbb{C}_1 \)

4. \( \text{Obj}(\mathbb{C}_2) = \text{Obj}(\mathbb{C}_1) = \text{Obj}(\mathbb{C}) \)

5. \( \text{Hom}(\mathbb{C}_2)(X, Y) = \text{Hom}(\mathbb{C}_1)(X, \Phi_1 Y) = \text{Hom}(\mathbb{C})(\Phi X, \Phi Y) \)

6. \( \forall f^+ : X \to Y \ g^+ : Y \to Z, \ g^+ \circ(\mathbb{C}_2) f^+ = (\mu_Y \circ(\mathbb{C}_1) \Phi_1 g^* \circ(\mathbb{C}_1) f^*) \in \mathbb{C}_1. \)

9. return **Any impure morphism** \( f^{(2)} : X \to Y \in \text{Hom}(\mathbb{C}_2) \) **is interpreted as**
    \( f_0 : X \times S \to Y \times S \in \text{Hom}(\mathbb{C}) \) **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:**

\[
\text{Inductive Exp : Type} \rightarrow \text{Type} :=
\]

- \(\text{const} : \forall A, A \rightarrow \text{Exp} A\)
- \(\text{loc} : \text{Loc} \rightarrow \text{Exp} Z\)
- \(\text{apply} : \forall A B, (A \rightarrow B) \rightarrow \text{Exp} A \rightarrow \text{Exp} B\)
- \(\text{pairExp} : \forall A B, \text{Exp} A \rightarrow \text{Exp} B \rightarrow \text{Exp} (A \times B)\).

**Fixpoint** \(\text{defExp} A (e : \text{Exp} A) : \text{term} A \text{ unit} :=\)

\[
\text{match} e \text{ with}
\]

- \(\text{const} Z n \Rightarrow \text{constant} n\)
- \(\text{loc} x \Rightarrow \text{lookup} x\)
- \(\text{apply} \_ \_ f x \Rightarrow \text{tpure} f o (\text{defExp} x)\)
- \(\text{pairExp} \_ \_ x y \Rightarrow \text{pair} (\text{defExp} x) (\text{defExp} y)\)

end.
IMP semantics over decorated logic

**Arithmetic and boolean expression declaration:**

**Inductive** `Exp : Type → Type :=

\[const : \forall A, A \rightarrow Exp A \]

\[loc : Loc \rightarrow Exp Z\]

\[apply : \forall A B, (A \rightarrow B) \rightarrow Exp A \rightarrow Exp B\]

\[pairExp : \forall A B, Exp A \rightarrow Exp B \rightarrow Exp (A \times B)\].

**Fixpoint** `defExp A (e : Exp A) : term A unit :=

\[
\text{match } e \text{ with}
\begin{align*}
\text{| const Z n } &\Rightarrow \text{constant n} \\
\text{| loc x } &\Rightarrow \text{lookup x} \\
\text{| apply \( f \) x } &\Rightarrow \text{tpure } f \circ (\text{defExp x}) \\
\text{| pairExp \( x \) y } &\Rightarrow \text{pair } (\text{defExp x}) (\text{defExp y})
\end{align*}
\text{end.}
\]

**A Coq side pure function: add**

**Definition** `add (p: Z × Z): Z :=

\[
\text{match } p \text{ with}
\begin{align*}
\text{| (x, y) } &\Rightarrow x + y
\end{align*}
\text{end.}
\]

**Pure Coq functions in operation**

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

**Check** `defExp(apply add (pairExp (const 30) (const 40))) =

\[
\text{tpure } add \circ (\text{pair } (\text{constant } 30) (\text{constant } 40)) : \text{term } Z \text{ unit.}
\]

\[
\text{IMP\_IntAop: } \forall (p q: Z) (f: Z \times Z \rightarrow Z),
\text{tpure } f \circ (\text{pair } (@\text{constant } Z p) (@\text{constant } Z q)) == \text{(constant } (f(p,q)))
\]

\[
\text{29 / 36}
\]
Appendix

IMP semantics over decorated logic cont’d

Command declaration

```ml
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
```

Fixpoint defCommand (c: Command): (term unit unit) :=
match c
| 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)
| loopdec (passbool o (defExp b)) (defCommand c4) o (defCommand c4) o (passbool o (defExp b))
Appendix

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

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

\[
\{\{x := n ; \\
   \text{if } (x \geq m) \ldots \} \}
\]

Left hand side diagram is strongly equal to the right hand side one.
Conditionals in decorated settings

\[
\{ \{ x := n ; \text{ if (} x >= m \text{) then } f \text{ else } g \} \}
\]

If \( b \) is \texttt{true} then \( f \) else \( g \) is executed.
Loops in decorated settings

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

Replace \text{loopdec} with the whole diagram as long as \( b \) is \text{ttrue}. Quit the loop through \( id_1 \) when it becomes \text{ffalse}. 
throw & try-catch in decorated settings

\{\{\text{throw } e\}\}\}.

\begin{center}
\begin{tikzpicture}[->,thick]
  \node (1) at (0,0) {$\bot$};
  \node (2) at (2,0) {$\emptyset$};
  \node (3) at (4,0) {$\bot$};
  \draw (1) to node {$\text{tag}_e$} (2);
  \draw (2) to node [swap] {$1$} (3);
\end{tikzpicture}
\end{center}

throwing an exception of name $e$.

\{\{\text{try } f \text{ catch } e \Rightarrow g\}\}\}.

\begin{center}
\begin{tikzpicture}[->,thick]
  \node (1) at (0,0) {$\bot$};
  \node (2) at (2,0) {$\bot + \emptyset$};
  \node (3) at (4,0) {$[id|k]$};
  \node (4) at (5,0) {$\bot$};
  \node (5) at (7,0) {$\bot$};
  \node (6) at (9,0) {$\bot$};
  \draw (1) to node {$f$} (2);
  \draw (2) to node [swap] {$id_1$} (1);
  \draw (2) to node [swap] {$id_1$} (3);
  \draw (3) to node [swap] {$id_0$} (2);
  \draw (3) to node [swap] {$k$} (4);
  \draw (4) to node {$\text{untag}_e$} (5);
  \draw (5) to node [swap] {$g$} (6);
\end{tikzpicture}
\end{center}

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:

\[(\text{SKIP})\{\phi\} \text{skip} \{\phi\} \quad (\text{ASSIGN})\{\phi[a/x]\} x := a \{\phi\}\]

\[(\text{SEQ}) \{\phi\} c_1 \{\psi\} \quad \{\psi\} c_2 \{\sigma\} \quad \{\phi\} c_1 ; c_2 \{\sigma\}\]

\[(\text{COND}) b \wedge \phi \quad c \{\psi\} \quad \neg b \wedge \phi \quad d \{\psi\} \quad \{\phi\} \text{if } b \text{ then } c \text{ else } d \{\psi\}\]

\[(\text{LOOP}) b \wedge \phi \quad c \{\phi\} \quad \{\phi\} \text{while } b \text{ do } c \{\phi \wedge \neg b\}\]

\[(\text{CONSEQ}) \phi \rightarrow \phi' \quad \{\phi'\} c \{\psi'\} \quad \psi' \rightarrow \psi \quad \{\phi\} c \{\psi\}\]

Hoare logic for exceptions: See [lecture notes](#) by Claude Marché