Program certification with computational effects

Burak Ekici*

j.w.w. Jean-Guillaume Dumas*, Dominique Duval*, Damien Pous†

*LJK, University Joseph Fourier, Grenoble
†LIP, ENS-Lyon

November 5, 2014

JNCF'14, Marseille-France

Contents

1. Dynamic evaluation through exceptions
2. Proofs with side effects
3. Coq in play

Contents

1. Dynamic evaluation through exceptions
2. Proofs with side effects
3. Coq in play

Dynamic evaluation

Dynamic evaluation := automatic case distinction process...

Dynamic evaluation

Dynamic evaluation := automatic case distinction process...

- Code re-usability:

Dynamic evaluation

Dynamic evaluation := automatic case distinction process...

- Code re-usability:

  E.g. Reusing codes made for `fields` over `rings`.

Dynamic evaluation

Dynamic evaluation := automatic case distinction process...

- Code re-usability:

  E.g. Reusing codes made for `fields` over `rings`.

  Gaussian elimination modulo `prime p` for Gaussian elimination modulo `composite m`.

Dynamic evaluation for modular Gaussian Elimination

- pivoting (α): not only `non-zero` but also `invertible`
- if any α is `non-zero` but `non-invertible` then

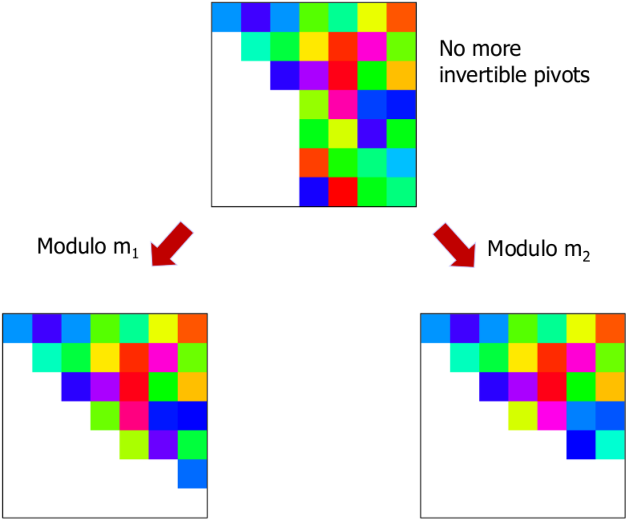
  `SPLIT` the computation for modulo $\mathtt{m}_1$ and $\mathtt{m}_2$ by `gcd` computation.

  [$\mathtt{m} = \mathtt{m}_1 \cdot \mathtt{m}_2$] & [$\mathtt{m}_1$ and $\mathtt{m}_2$ are gcd-free]

  [α is `invertible` modulo $\mathtt{m}_1$] & [α is `zero` modulo $\mathtt{m}_2$]

  1. Gaussian elimination modulo m_1
  2. Gaussian elimination modulo m_2

Dynamic evaluation for modular Gaussian Elimination



Dynamic evaluation via exceptions

- arithmetic level exception: preventing zero divisors also

```
inline Integer invmod(const Integer& a, const Integer& m) {
      Integer gcd,u,v;
      ExtendedEuclideanAlgorithm(gcd,u,v,a,m);
      if (gcd != 1) throw ZmzInvByZero(gcd);
      return v>0?v:v=+m;
}
```

- exception at split location

```
try {
      invpiv = zmz(1)/A[k][k];
} catch (ZmzInvByZero e) {
      throw GaussNonInvPivot(e.getGcd(), k, currentrank);
}
```

Dynamic evaluation via exceptions 2/2

- deal with split: recursive continuation

```
try { // in place modifications of lower n-k part of matrix A
        int rank = gaussrank(A,k);
        cout << rank: << rank + upperrank << modulo << m;
} catch (GaussNonInvPivot e) {
        // recursive continuation modulo m1 AND modulo m2
        // at current step
}
```

Contents

1. Dynamic evaluation through exceptions
2. Proofs with side effects
3. Coq in play

Programs with side effects 1/2

Side effect := the mismatch between syntax and semantics...

E.g. The *exceptions* effect:

Considering the `exception thrower`:

```
float function(int a) throw(Exception) { ... };
```

⇒ Syntactically;

function: int → float

⇒ w.r.t. an interpretation (denotational semantics);

function: int → float + Exception

E.g. The *state* effect:

Considering the `state modifier`:

```
class S {
    float method (int a) { ... };
}

...

S t; t.method(10);
```

⇒ Syntactically;

method: int $\to$ float

⇒ w.r.t. an interpretation (denotational semantics);

method: int $\times$ S $\to$ float $\times$ S

Decorated logic [Dominguez & Duval'08]

tools for modeling computations with effects:

- `monads`: [Moggi'91]
- `decorated logic`: based on the framework by [Dominguez & Duval'08]
  - provides `equivalence proofs` among `programs with effects`

Decorated logic [Dominguez & Duval'08]

tools for modeling computations with effects:

- `monads`: [Moggi'91]
- `decorated logic`: based on the framework by [Dominguez & Duval'08]
  - provides `equivalence proofs` among `programs with effects`

⇒ `Equivalence proofs` are aimed to be verified by Coq.

Decorated logic: exceptions 1/2

$f^{(0)}$	:	$X \to Y$	pure
$f^{(1)}$	:	$X \to Y$	thrower/propagator
$f^{(2)}$	:	$X \to Y$	catcher

specify
the decoration ↗

↘ explain
the decoration

f	:	$X \to Y$

f	:	$X \to Y$
f	:	$X \to Y+E$
f	:	$X+E \to Y+E$

⇒ Ease of composition: exceptional behaviors are kept implicit.

I.e.,

Given $f^{(2)} : X \to Y$ and $g^{(1)} : Y \to Z$, $(g \circ f)^{(2)} : X \to Z$

Decorated logic: exceptions 2/2

- strong equality (on ordinary and exceptional arguments) $f \equiv g$
- weak equality (on ordinary arguments only) $f \sim g$

$$\boxed{\begin{array}{lcl} f \equiv g & : & X \to Y \\ f \sim g & : & X \to Y \end{array}}$$

specify the decoration ↗

↘ explain the decoration

$$\boxed{f = g \quad : \quad X \to Y}$$

$$\boxed{\begin{array}{lcl} f = g & : & X{+}E \to Y{+}E \\ f \circ \mathtt{inl}_X = g \circ \mathtt{inl}_X & : & X \to Y{+}E \end{array}}$$

[$\mathtt{inl}_X$ is the inclusion of X into $X{+}E$]

⇒ More precise equational proofs of programs: w.r.t. effects and ordinary cases.

Core exceptional operations: tag/untag

$$\mathtt{tag}_t \; : P_t \to \mathbb{0}$$
$$\mathtt{untag}_t : \mathbb{0} \to P_t$$

ordinary value (normal)		exceptional value (abrupt)
a	$\xrightarrow{\mathtt{tag}_t}$	$\boxed{a}_t$
a	$\xleftarrow{\mathtt{untag}_t}$	$\boxed{a}_t$

$\Rightarrow$ `throwing` and `catching` exceptions := core operations + pattern matching.

throwing & handling exceptions

⇒ Throwing an exception := $\mathtt{tag}_t$ and some glue for the continuation.

$$\mathtt{throw}^{(1)}_{t,Y} := [\ ]^{(0)}_Y \circ \mathtt{tag}^{(1)}_t : P_t \to \mathbb{0} \to \mathbb{0} + Y \cong Y \quad : P_t \to Y$$

⇒ Exception handling := $\mathtt{untag}_t$ with pattern matching.

Considering the handler $g^{(1)} : P_t \to Y$:

$$\mathtt{catch}(t \Rightarrow g)^{(2)} := [\mathtt{id}^{(0)}_Y \mid g^{(1)} \circ \mathtt{untag}^{(2)}_t] \qquad : Y + \mathbb{0} \cong Y \to Y$$

try-catch block

$\Rightarrow$ $\mathtt{try} - \mathtt{catch}$ block can be expressed by compositions of decorated terms:

For any $\mathtt{f}^{(1)} : \mathtt{X} \to \mathtt{Y}$:

$\mathtt{try\{f\}\ catch(t \Rightarrow g)}^{(1)} :=$

$$\downarrow ([\mathtt{id}_{\mathtt{Y}}^{(0)} \mid \mathtt{g}^{(1)} \circ \mathtt{untag}_{\mathtt{t}}^{(2)}] \circ \mathtt{f}^{(1)}) : \mathtt{X} \to \mathtt{Y} \cong \mathtt{Y} + \mathbb{0} \cong \mathtt{Y} \to \mathtt{Y}$$

$\Rightarrow$ $\mathtt{try}$ bounds the scope of $\mathtt{catch}$

Decorated logic: exceptions - rules

The given logic is enriched with some number of rules:

- Conversion rules

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

- Equivalence rules
- Rules on monadic equational logic
- Categorical coproduct rules
- Observational properties: `tag` & `untag`

$$(ax_1)\frac{\mathtt{t}: \mathtt{Excn}}{\mathtt{untag}_{\mathtt{t}}^{(2)} \circ \mathtt{tag}_{\mathtt{t}}^{(1)} \sim \mathtt{id}_{\mathtt{P}_{\mathtt{t}}}{}^{(0)}} \qquad (ax_2)\frac{\mathtt{t}, \mathtt{r}: \mathtt{Excn} \quad \mathtt{t} \neq \mathtt{r}}{\mathtt{untag}_{\mathtt{r}}^{(2)} \circ \mathtt{tag}_{\mathtt{t}}^{(1)} \sim [\,]_{\mathtt{P}_{\mathtt{r}}}^{(0)} \circ \mathtt{tag}_{\mathtt{t}}^{(1)}}$$

Soundness of the inference system

Axioms/rules allow us to prove:

1. propagator propagates: $\mathtt{g}^{(1)} \circ [\,]_{\mathtt{X}}^{(0)} \equiv [\,]_{\mathtt{Y}}^{(0)}$
2. annihilation untag-tag: $\mathtt{tag}_{\mathtt{t}}^{(1)} \circ \mathtt{untag}_{\mathtt{t}}^{(2)} \equiv \mathtt{id}_{\mathbb{0}}^{(0)}$
3. annihilation catch-raise: $\mathtt{try\{f\}\ catch(t \Rightarrow throw_{t,Y})}^{(1)} \equiv \mathtt{f}^{(1)}$
4. commutation untag-untag: given $\mathtt{s} \neq \mathtt{t}$
   $(\mathtt{untag}_{\mathtt{t}}^{(2)} + \mathtt{id}_{\mathtt{s}}^{(0)}) \circ \mathtt{untag}_{\mathtt{s}}^{(2)} \equiv (\mathtt{id}_{\mathtt{t}}^{(0)} + \mathtt{untag}_{\mathtt{s}}^{(2)}) \circ \mathtt{untag}_{\mathtt{t}}^{(2)}$
5. commutation catch-catch: given $\mathtt{s} \neq \mathtt{t}$
   $\mathtt{try\{f\}\ catch(t \Rightarrow g \mid s \Rightarrow h)}^{(1)} \equiv \mathtt{try\{f\}\ catch(s \Rightarrow h \mid t \Rightarrow g)}^{(1)}$

Contents

1 Dynamic evaluation through exceptions

2 Proofs with side effects

3 Coq in play

Coq in one slide

Coq:

- *proof assistant*
- *strongly typed*, *purely functional* programming language
  - not Turing complete: non-termination avoided

Coq in one slide

Coq:

- *proof assistant*
- *strongly typed*, *purely functional* programming language
  - not Turing complete: non-termination avoided

⇒ Underlying type theory: Calculus of Inductive Constructions (CIC) [Coquand et al'89].

Coq in one slide

Coq:

- *proof assistant*
- *strongly typed*, *purely functional* programming language
  - not Turing complete: non-termination avoided

⇒ Underlying type theory: Calculus of Inductive Constructions (CIC) [Coquand et al'89].

CIC:

- extension to simply typed lambda calculus with
  - polymorphism: terms depending on types
  - type operators: types depending on types
  - dependent types: types depending on terms
  - inductive definitions
- `Type` predicativity (hierarchy): to avoid Russell-like paradoxes.

Coq formalization of exceptions 1/3

Every function f : X → Y becomes f : term Y X in decorated settings which is inductively defined:

```
Inductive term: Type → Type → Type :=
  | comp: ∀ {X Y Z: Type}, term X Y → term Y Z → term X Z
  | copair: ∀ {X Y Z: Type}, term Z X → term Z Y → term Z (X+Y)
  | tpure: ∀ {X Y: Type}, (X → Y) → term Y X
  | tag: ∀ t:Exn, term Empty_set (P t)
  | untag: ∀ t:Exn, term (P t) Empty_set.
```

and `decorated` as follows:

0 = `pure`
1 = `propagator`
2 = `catcher`

```
Inductive kind := pure | propagator | catcher.
```

```
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_copair: ∀ k X Y Z (f: term Z X) (g: term Z Y), is k f → is k g →
                                                   is k (copair f g)
 | is_tag: ∀ t, is propagator (tag t)
 | is_untag: ∀ t, is catcher (untag t)
 | is_pure_propagator: ∀ X Y (f: term X Y), is pure f → is propagator f
 | is_propagator_catcher: ∀ X Y (f: term X Y), is propagator f → is catcher f.
```

`rules` are inductively defined, too:

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

Inductive strong: ∀ X Y, relation (term X Y) :=
    | effect_rule: ∀ X Y (f g: term Y X), f o [ ] == g o [ ] → f ~ g → f == g
                    ⋮
with weak: ∀ X Y, relation (term X Y) :=
    | ax_1: ∀ t, untag t o tag t ~ id
                    ⋮
```

IMP-STATES-EXCEPTIONS: the library

`IMP+EXC` is an imperative language enriched with exceptions:

IMP-STATES-EXCEPTIONS: the library

IMP+EXC is an imperative language enriched with exceptions:

IMP+EXC Syntax:

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

IMP-STATES-EXCEPTIONS: the library

`IMP+EXC` is an imperative language enriched with exceptions:

`IMP+EXC Syntax:`

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

⇒ Operational semantics of `IMP+EXC`: `IMP-STATES-EXCEPTIONS`

▸ source code (IMP-STATES-EXCEPTIONS)

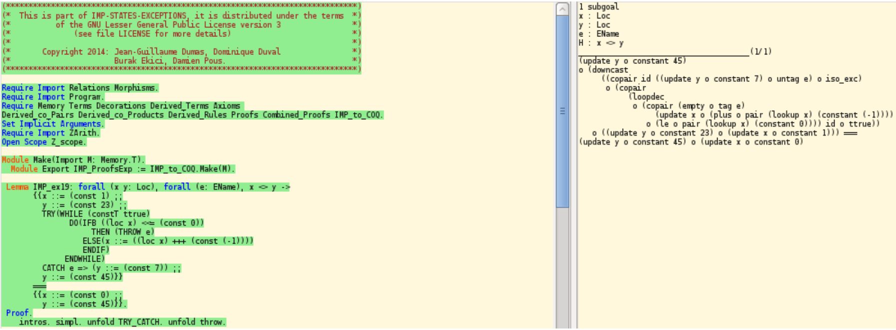
Verified programs 1/2

E.g.,

```
prog_1 = (
  var x, y ;
  x := 1 ; y := 23 ;
  try(
    while(tt) do (
      if(x <= 0)
      then(throw e)
      else(x := x - 1)
    )
  )
  catch e => (y := 7) ;
  y := 45 ;
) .
```

==

```
prog_2 = (
 var x, y ;
 x := 0 ; y := 45 ;
) .
```



Verified programs 2/2

```
prog_3= (
  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_4= (
  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 ;
).
```

Verified programs 2/2

```
prog_3= (
  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_4= (
  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.

Consider:

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

self-evaluation + open questions

+ `aspect`: having *verified proofs* of programs with effects

- `aspect`: not so good benchmarks

  ⇒ different orders of magnitude to have better performances

self-evaluation + open questions

+ `aspect`: having *verified proofs* of programs with effects

- `aspect`: not so good benchmarks

  ⇒ different orders of magnitude to have better performances

- `aspect`: proofs of real-valued programs (where all variables are initialized)

+ `aspect`: decorated logic w.r.t. *weak equality* corresponds to *Hoare logic* (formal system to reason about partial correctness of programs)

So far & future work

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

So far & future work

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

Future:

- Hilbert-Post Completeness proof for exceptions
- systematic way to compose effects + generalization

The end!

`Many thanks for your kind attention!`

`Questions?`