

Certification of programs with computational effects

Burak Ekici*

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

*LJK, University Joseph Fourier, France

†LIP, ENS-Lyon, France

Casys-Meff Seminars'14
Grenoble, France

November 20, 2014

Motivation

- proving program properties involving computational (side) effects:
 - State
 - Exceptions
- through decorated logic

Motivation

- proving program properties involving computational (side) effects:
 - State
 - Exceptions
- through decorated logic
- with related completeness proofs

Motivation

- proving program properties involving computational (side) effects:
 - State
 - Exceptions
- through decorated logic
- with related completeness proofs
 - Coq certification of mentioned proofs

Decorated logic

[Dominguez & Duval'08]

1991 Eugenio Moggi

- Monads to model effects



2008 Dominguez and Duval

- Decorated Logic: Cartesian Effect Categories

⇒ equivalence proofs among programs with effects

Decorated logic: exceptions - syntax

1/2

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

specify
the decoration



explain
the decoration



f	:	$X \rightarrow Y$
-----	---	-------------------

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

\Rightarrow Ease of composition: exceptional behaviors are kept implicit.

I.e.,

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

Decorated logic: exceptions - syntax

2/2

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

$$\begin{array}{l} f \equiv g : X \rightarrow Y \\ f \sim g : X \rightarrow Y \end{array}$$

specify
the decoration ↗

$$f = g : X \rightarrow Y$$

↘ explain
the decoration

$$\begin{array}{l} f = g : X+E \rightarrow Y+E \\ f \circ \text{inl}_X = g \circ \text{inl}_X : X \rightarrow Y+E \end{array}$$

inl_X is the inclusion of X into $X+E$

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

Decorated logic: exceptions - rules

The given logic is enriched with some number of rules:

- Conversion rules

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

- Equivalence rules
- Rules on monadic equational logic
- Categorical coproduct rules
- Observational properties of core exceptional operations

Soundness of the inference system

Such a formalization allows us to prove primitive properties of programs with exceptions:

- commutation catch-catch: given $s \neq t$

$$\text{try}\{f\} \text{ catch}(t \Rightarrow g \mid s \Rightarrow h)^{(1)} \equiv \text{try}\{f\} \text{ catch}(s \Rightarrow h \mid t \Rightarrow g)^{(1)}$$

Automatizing mathematics: λC

[Coquand et al'85]

1960 Curry-Howard Correspondence

- Proofs as programs



1985 Thierry Coquand & Gérard Huet

- A formal language to write proofs:
Calculus of Constructions (λC)



Coq

[Coquand et al'88]

Calculus of Constructions (λC):

- extension to λ^{\rightarrow} with
 - polymorphism, type operators and dependent types

Coq

[Coquand et al'88]

Calculus of Constructions (λC):

- extension to λ^{\rightarrow} with
 - polymorphism, type operators and dependent types

1988 Thierry Coquand & Christine Paulin

- λC + inductive definitions := Calculus of Inductive Constructions (CIC or Coq)

Coq

[Coquand et al'88]

Calculus of Constructions (λC):

- extension to λ^{\rightarrow} with
 - polymorphism, type operators and dependent types

1988 Thierry Coquand & Christine Paulin

- λC + inductive definitions := Calculus of Inductive Constructions (CIC or Coq)

2004 Bruno Barras & Hugo Herbelin

- Coq + Type predicativity \rightarrow to avoid Russell-like paradoxes := Coq 8.0.

Coq

[Coquand et al'88]

Calculus of Constructions (λC):

- extension to λ^{\rightarrow} with
 - polymorphism, type operators and dependent types

1988 Thierry Coquand & Christine Paulin

- λC + inductive definitions := Calculus of Inductive Constructions (CIC or Coq)

2004 Bruno Barras & Hugo Herbelin

- Coq + Type predicativity \rightarrow to avoid Russell-like paradoxes := Coq 8.0.

2014 Bertot & Courtieu & Filliâtre & Marché & Sozeau & ...

- Coq 8.4:
 - proof assistant
 - strongly typed, purely functional programming language
 - not Turing complete: non-termination avoided
 - Co-inductive definitions: to cope with infinite data structures.
I.e., streams

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:

Syntax:

```
aexp:  a1 a2 ::= ...
bexp:  b1 b2 ::= ...
cmd :  c1 c2 ::= skip | x := e | c1; c2 | if b then c1 else c2 |
      while b do c1 | throw exc | try c1 catch exc ⇒ c2
```


IMP-STATES-EXCEPTIONS: the library

IMP+EXC is an imperative language enriched with exceptions:

Syntax:

```
aexp:  a1 a2 ::= ...
bexp:  b1 b2 ::= ...
cmd :  c1 c2 ::= skip | x := e | c1; c2 | if b then c1 else c2 |
        while b do c1 | throw exc | try c1 catch exc ⇒ c2
```

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

[▶ source code](#) (IMP-STATES-EXCEPTIONS)

Verified programs

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 ;
) .

```

```

-----
|* This is part of IMP_STATES-EXCEPTIONS, it is distributed under the terms
|* of the GNU Lesser General Public License version 3
|* (see file LICENSE for more details)
|*
|* Copyright 2014 Jean-Guillaume Dumas, Dominique Duval
|* Burak Elci, Damien Fouz
|*
-----
Require Import Relations Morphisms.
Require Import Program.
Require Memory Terms Decorations Derived_Terms Baisons.
Derived_on_Pairs Derived_on_Products Derived_Rules Proofs Combined_Proofs IMP_to_COQ.
Set Implicit Arguments.
Require Import ZArith.
Open Scope Z_scope.

Module Make(Import M: Memory.T).
Module Export IMP_ProofsExp := IMP_to_COQ.Make(M).

Lemma IMP_exp19: forall (x y : Loc), forall (e : EName), x << y =>
  ((x := (const 1)) ;
   y := (const 23)) ;
  TRY(WHILE (const ttrue)
    DO(IFB ((loc x) <= (const 0))
      THEN (THROW e)
      ELSE x := ((loc x) ++ (const (-1)))
    ENDIF)
  ENDWHILE)
  CATCH e => (y := (const 7)) ;
  y := (const 45)) ;
  ((x := (const 0)) ;
   y := (const 45)) ;
Proof.
  intros. simpl. unfold TRY_CATCH. unfold throw.

```

```

1 subgoal
x : Loc
y : Loc
e : EName
H : x << y
----- (1/1)
(update y o constant 45)
o (downcast
  (copair_id ((update y o constant 7) o untag e) o iso_exc)
  o (copair
    (loopdec
      o (copair (empty o tag e)
        (update x o (plus o pair (lookup x) (constant (-1))))
        o (le o pair (lookup x) (constant 0)))) id o ttrue))
    o ((update y o constant 23) o (update x o constant 1))) ==
(update y o constant 45) o (update x o constant 0)

```

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?