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

$\Rightarrow$ `equivalence proofs among programs with effects`

Decorated logic: exceptions - syntax 1/2

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

specify
the decoration $\nearrow$

$\searrow$ explain
the decoration

$\boxed{\mathtt{f} \quad : \quad \mathtt{X} \to \mathtt{Y}}$

$\mathtt{f}$	:	$\mathtt{X} \to \mathtt{Y}$
$\mathtt{f}$	:	$\mathtt{X} \to \mathtt{Y+E}$
$\mathtt{f}$	:	$\mathtt{X+E} \to \mathtt{Y+E}$

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

I.e.,

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

Decorated logic: exceptions - syntax 2/2

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

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

specify
the decoration $\nearrow$

$\searrow$ 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 $X{+}E$

$\Rightarrow$ 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

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

- Equivalence rules
- Rules on monadic equational logic
- Categorical coproduct rules
- Observational properties 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 $\mathtt{s} \neq \mathtt{t}$

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

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 $\lambda^{\rightarrow}$ with
  - polymorphism, type operators and dependent types

Coq [Coquand et al'88]

Calculus of Constructions ($\lambda\mathtt{C}$):

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

1988 Thierry Coquand & Christine Paulin

- $\lambda\mathtt{C}$ + inductive definitions := Calculus of Inductive Constructions (CIC or Coq)

Coq [Coquand et al'88]

Calculus of Constructions (λC):

- extension to $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$ 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:

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

IMP-STATES-EXCEPTIONS: the library

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

`Syntax:`

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

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

▸ source code (IMP-STATES-EXCEPTIONS)

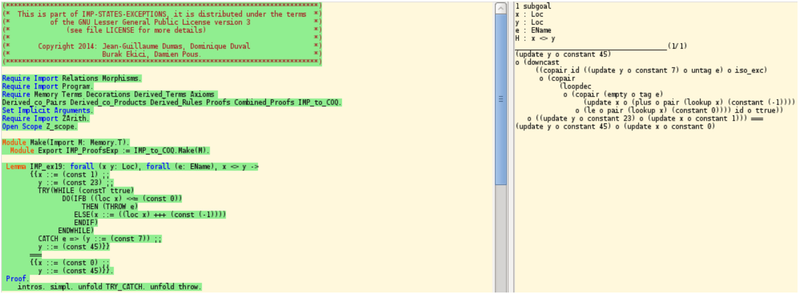
Verified programs

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 ;
) .
```



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?