The art of lying, with the proof assistant Coq

Dominique Duval, with
Jean-Guillaume Dumas, Burak Ekici, Jean-Claude Reynaud
and Damien Pous

IFIP WG1.3, Sinaia Meeting, September 3, 2014

"the wise thing is for us diligently to train ourselves to lie thoughtfully, judiciously..."
On the Decay of the Art of Lying, Mark Twain, 1880.

"A lie-to-children is a simplified explanation of technical or complex subjects as a teaching method for children..."
Lie-to-children, Wikipedia, 2014.

Outline

Lying in Computer Algebra: dynamic evaluation

Re-use code made for fields... with rings

Example:

- Write code for Gaussian elimination modulo p,
  for a prime number p
- Use this code for Gaussian elimination modulo m,
  for any natural number m (including m composite)

Dynamic evaluation is an automatic case distinction process

Dynamic evaluation for Gaussian elimination mod. m

- Use pivot, as long as they are invertible (not only non-zero)
- In case of non-zero but non-invertible pivot SPLIT the computation in two parts with $m = m1 \times m2$, where $m1$ and $m2$ are coprime
  1. Remaining Gaussian elimination modulo $m1$
  2. Remaining Gaussian elimination modulo $m2$

Outline

Lying in Programming: computational effects

"programs of type B with a parameter of type A ought to be interpreted by morphisms from A to TB..."
Notions of computation and monads, E. Moggi, 1991

Example:

- Write code as if division by zero may never occur
- Add exceptions for dealing with occasional division by zero

For exceptions let $TX = X + E$.

Dynamic evaluation via exceptions

1. Add an exception at the arithmetic level
   to prevent against zero divisors
   ```
   ...if (gcd != 1) throw ZmzInvByZero(gcd);...
   ```
2. Add an exception at the SPLIT location
   to allow for recursive continuation
   ```
   try {...
   } catch (ZmzInvByZero e) {
      throw GaussNonInvPivot(...);
   }
   ```
3. Deal with the SPLIT
   ```
   try {...
   } catch (GaussNonInvPivot e) {...
   // recursive continuation modulo m1 AND modulo m2
   }
   ```

Outline

Motivation

- verify properties of programs involving computational effects such as state and exceptions
- keep close to the syntax: do not make the effects explicit, but use "decorations" instead (like Moggi's *values* and *computations*)
- and develop related Coq libraries for such proofs

Decorations for states

For each location T, let V_T be the type of values that can be stored in T. Terms are classified by decorations:

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

There are two kinds of equations:

- strong equality (result and effect) $f \equiv g$
- weak equality (result only) $f \sim g$

Equational rules are decorated accordingly...

Decorations for exceptions 1: by DUALITY

For each exception name T, let V_T be the type of parameters for exceptions of name T. Terms are classified by decorations:

- pure: $\mathtt{id}^{(0)}_{\mathtt{V_T}} : V_T \to V_T$,
- propagators: $\mathtt{tag}^{(1)}_{\mathtt{T}} : V_T \to \mathbb{0}$
- catchers: $\mathtt{untag}^{(2)}_{\mathtt{T}} : \mathbb{0} \to V_T$

There are two kinds of equations:

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

Equational rules are decorated accordingly...

Decorations for exceptions 2

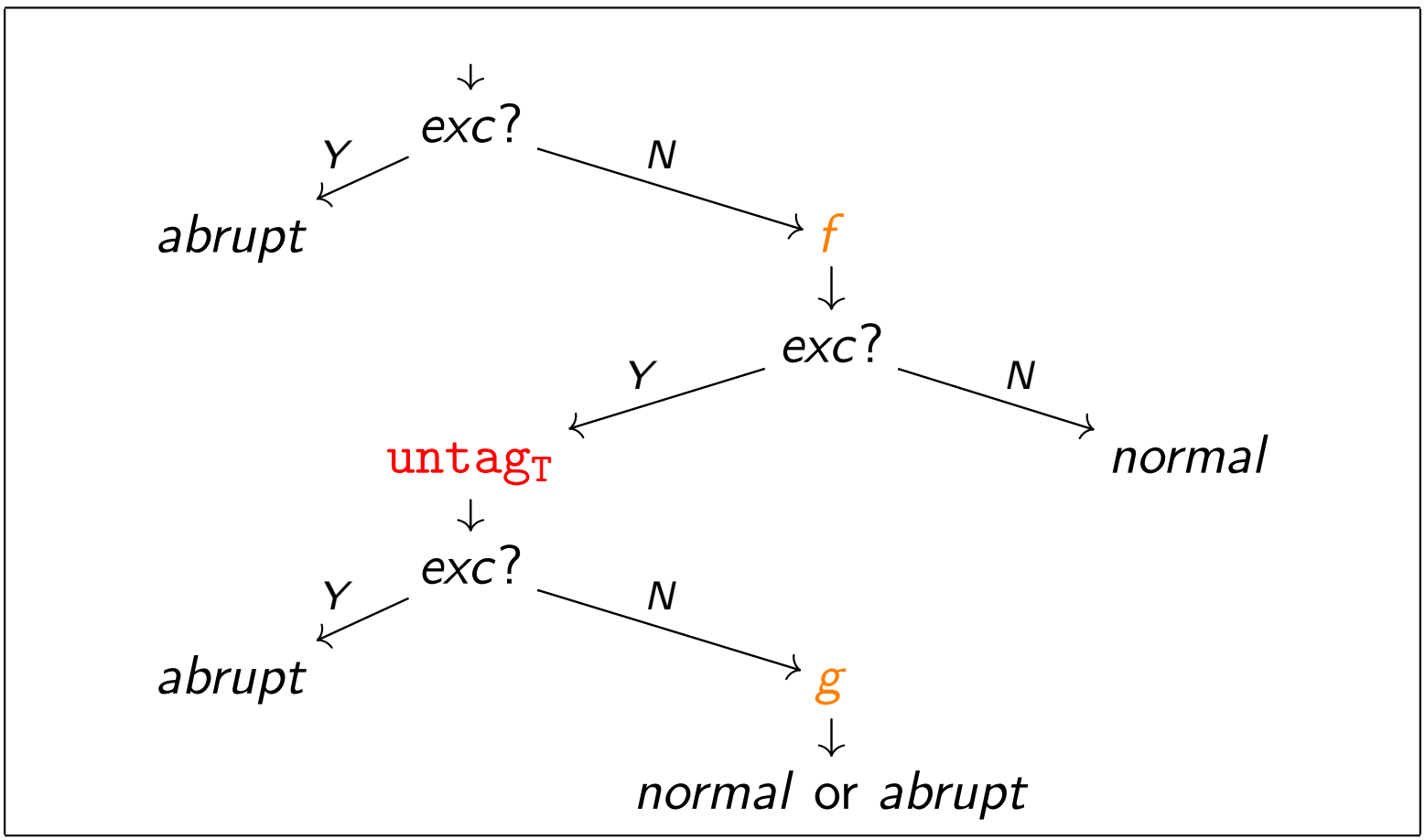
- Private language (dual to states): $\mathtt{tag}_{\mathtt{T}}^{(1)} : V_T \to \mathbb{0}$ and $\mathtt{untag}_{\mathtt{T}}^{(2)} : \mathbb{0} \to V_T$

ordinary value		exceptional value
a	$\xrightarrow{\mathtt{tag}_T}$	$\boxed{a}_T$
a	$\xleftarrow{\mathtt{untag}_T}$	$\boxed{a}_T$

- Public language (without any catcher):
  $\mathtt{throw}_{\mathtt{T},\mathtt{B}} : V_T \to B$
  and $\mathtt{try(f)\ catch(T \Rightarrow g)}_{T,A,B} : A \to B$
  for any $f : A \to B$ and $g : V_T \to B$

Decorations for exceptions 3

$\texttt{try(f) catch(T} \Rightarrow \texttt{g)}_{T,A,B} : A \to B$
for any $f : A \to B$ and $g : V_T \to B$



Decorated proofs in Coq

- Decorated proofs for states
- Decorated proofs for exceptions
- Decorated proofs for the IMP language (states)
- Decorated proofs for the IMP+EXC language (states and exceptions)

Coq for states

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).
```

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.
```

Proofs: basic properties

- For states: the usual equations, e.g.
  - Annihilation lookup-update
  - Commutation update-update
- For exceptions: the dual equations and their public version:
  - Annihilation untag-tag
  - Annihilation catch-raise
  - Commutation untag-untag
  - Commutation catch-catch

Proofs: Hilbert-Post completeness

With respect to the decorated logic for states or exceptions:

- A theory T is consistent if it does not contain all equations.
- A theory T' is a pure extension of T if it is generated by T and by equations between pure terms.
- A theory T' is a proper extension of T if it contains T and is not a pure extension of T.
- T is Hilbert-Post complete if is consistent and has no consistent proper extension.

Theorem.
The core language for exceptions is Hilbert-Post complete.

Proof for states.
Every equation between modifiers or accessors is equivalent to several equations between pure terms ...

Proof for exceptions (dual).
Every equation between catchers or propagators is equivalent to several equations between pure terms ...

From IMP to decorated logic

IMP syntax:

Variables:

$$v ::= X \mid Y \mid \ldots$$

Arithmetic expressions:

$$a ::= 0 \mid 1 \mid \ldots \mid v \mid a + a \mid \ldots$$

Boolean expressions:

$$b ::= \texttt{true} \mid \texttt{false} \mid b \wedge b \mid \cdots \mid a = a \mid \ldots$$

Commands:

$$c ::= \texttt{skip} \mid v := a \mid c; c \mid \texttt{if } b \texttt{ then } c \texttt{ else } c \mid \texttt{while } b \texttt{ do } c$$

For states:

Expressions are accessors (read-only)

Commands are modifiers (read-write)

From IMP+EXC to decorated logic

IMP+EXC syntax:
Commands:

$$c ::= \texttt{throw}\ exc \mid \texttt{try}\ c\ \texttt{catch}\ exc \Rightarrow c$$

For exceptions:
These commands are propagators

Proofs: Dynamic evaluation

Compute the rank of a matrix modulo some composite number, and check that it returns the required value.

Because of the limitations of IMP+EXC (no kind of function!) this has been done only for 2×2 matrices...

Work in progress!

Future work

- From IMP to (part of) C
- Hoare logic with decorations
- Composition of effects

Thanks for your attention!