Certified proofs in programs involving exceptions

Jean-Guillaume Dumas
with D. Duval, B. Ekici, J.-C. Reynaud

Université de Grenoble

Laboratoire Jean Kuntzmann
Applied Mathematics and Computer Science Department

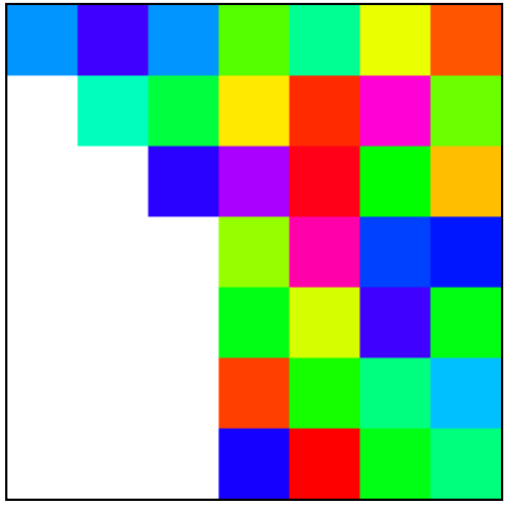
Dynamic Evaluation (D5) for modular Gaussian Elimination

- Re-use code made for fields … with rings
  - Example: Gaussian elimination modulo p, for p a prime
  - Used for: Gaussian elimination modulo m, m composite

- Dynamic evaluation
  - Use pivot, as long as they are invertible (not only non-zero)
  - In case of non-zero but non-invertible pivot a
  - SPLIT the computation in two parts with $m = m_1 \cdot m_2$
    1. Remaining Gaussian elimination modulo m_1
    2. Remaining Gaussian elimination modulo m_2

    ⇨ m_1 and m_2 are gcd-free & $\gcd(a,m_1) = 1$

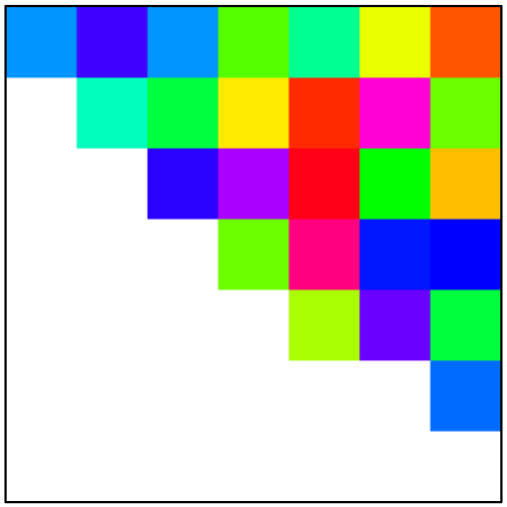
Dynamic Evaluation (D5) for modular Gaussian Elimination

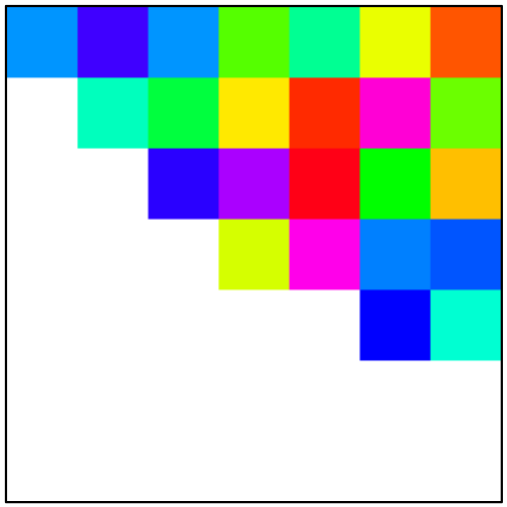


No more invertible pivots

Modulo m_1

Modulo m_2





Dynamic evaluation via exceptions 1/2

1. Add an exception at the arithmetic level

```
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 u>0?u:u+=m;
}
```

2. Add an exception at the SPLIT location

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

Dynamic evaluation via exceptions 2/2

3. Deal with the SPLIT

```
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
     ⋮ (just 6 more SLOC)
}
```

⇨ Low-key intrusiveness, exception at two levels:

- ⇨ At arithmetic level: prevents other unforeseen zero divisors
- ⇨ At Gaussian elimination level: allows for recursive continuation

Proofs involving side-effects

- Proving equivalence of programs
  - Arithmetic, If-Then-Else, Loops, etc.
  - Side-effects: mismatch syntax <--> semantics
  - Exception effect
    - `int f(float)` signature is not `f: float` $\rightarrow$ `int`,
    - it is `f: float` $\rightarrow$ `int+Exceptions`
  - see B. Ekici talk, Friday 11am, for other effects and combinations of effects …

Decorated logic for exceptions

- Syntax: $f: X \to Y$
- Denotation
  - f is pure if $[\![f]\!]: [\![X]\!] \to [\![Y]\!]$
  - f may raise exceptions if $[\![f]\!]: [\![X]\!] \to [\![Y]\!]+E$
  - f may catch exceptions if $[\![f]\!]: [\![X]\!]+E \to [\![Y]\!]+E$
- Decoration
  - $f^{(0)}$ is pure
  - $f^{(1)}$ may raise exceptions ($f^{(1)}$ is called a propagator)
  - $f^{(2)}$ may catch exceptions ($f^{(2)}$ is called a catcher)

Proofs at the decorated level: 2 stages

$$f^{(0)} : X \to Y$$
$$f^{(1)} : X \to Y$$
$$f^{(2)} : X \to Y$$

1) At a syntactic level

$$f : X \to Y$$

2) Effect-wise:
Take decorations into account

⇨ Correspond to explicit proof

$$[\![f]\!] : [\![X]\!] \to [\![Y]\!]$$
$$[\![f]\!] : [\![X]\!] \to [\![Y]\!] + E$$
$$[\![f]\!] : [\![X]\!] + E \to [\![Y]\!] + E$$

- Need also to decorate equations
  - $f \equiv g$ iff both results and effects are the same
  - $f \sim g$ if results are the same but effect can be different

Creating and returning from Exceptions

- Core operations
  - For an exception of type E with parameters of type T

ordinary value (normal)		exceptional value (abrupt)
a	$\xrightarrow{tag_T}$	$\boxed{a}_T$
a	$\xleftarrow{untag_T}$	$\boxed{a}_T$

  - $[\![\mathrm{tag}_T]\!] : [\![T]\!] \to [\![E]\!]$, i.e., $\mathrm{tag}_T{}^{(1)} : T \to \mathbb{O}$
  - $[\![\mathrm{untag}_T]\!] : [\![E]\!] \to [\![T]\!] + [\![E]\!]$, i.e., $\mathrm{untag}_T{}^{(2)} : \mathbb{O} \to T$

- Key properties
  - ($[\,]_X$ is inclusion of $\mathbb{O}$ into $\mathbb{O}+X$, that is $[\![[\,]_X]\!]$ is inclusion of E into $E+[\![X]\!]$)

$$\frac{T \in Exc}{untag_T \circ tag_T \sim id_T} \qquad \frac{R, T \in Exc \quad R \neq T}{untag_T \circ tag_R \sim [\,]_T \circ tag_R}$$

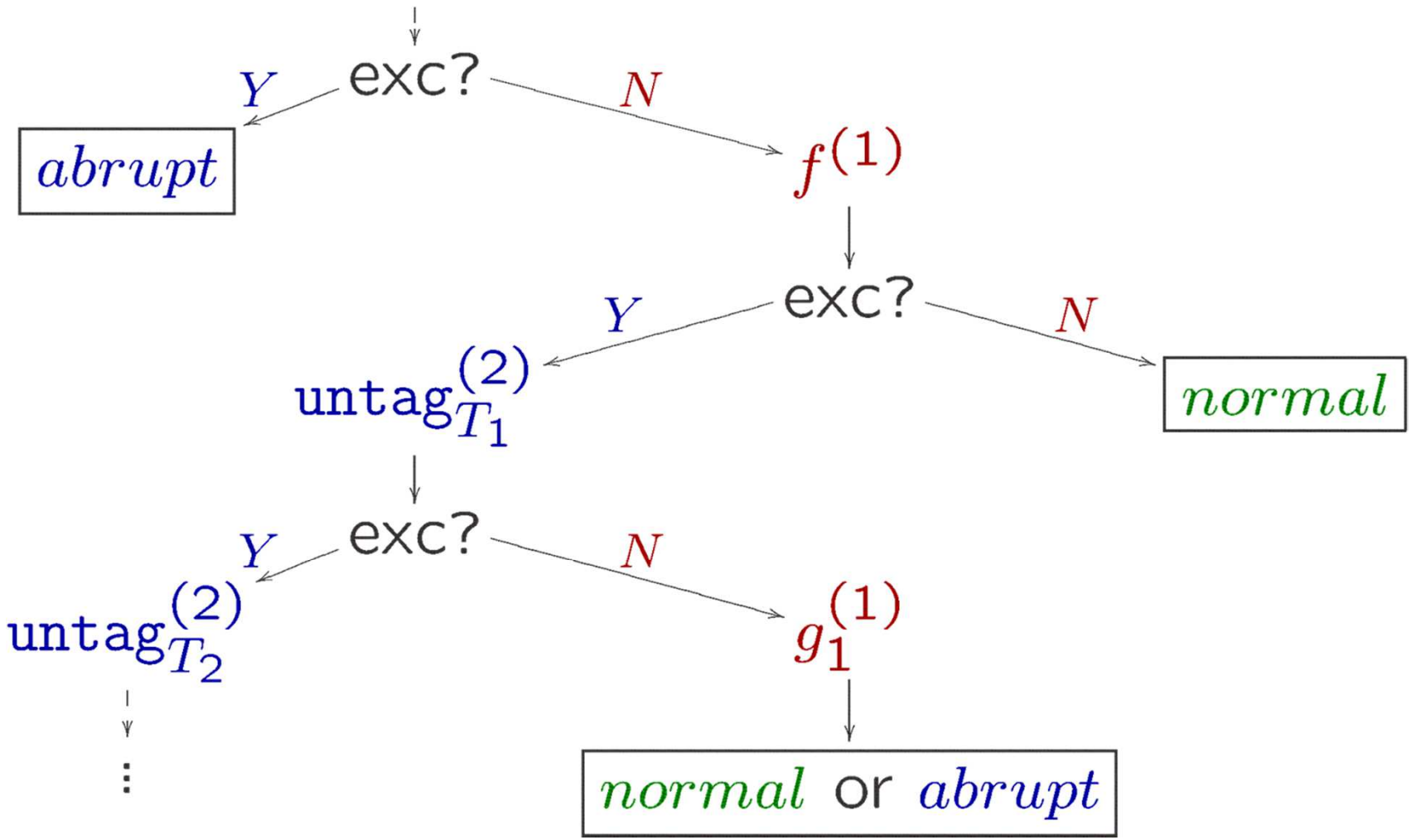
Throwing and handling exceptions

- $\mathrm{throw}_{T,Y}: T \to Y$ within $f: X \to Y$
  - throws an exception of parameter T within f returning Y
  - so define: $\mathrm{throw}_{T,Y} = [\,]_Y \circ \mathrm{tag}_T : T \to \mathbb{O} \to \mathbb{O}+Y \cong Y$

- Pattern matching for handling exceptions
  - For a handler $g: T \to Y$
  - $\mathrm{catch}(T \Rightarrow g) = [\mathrm{id}_Y \mid g \circ \mathrm{untag}_T] : \mathbb{O}+Y \cong Y \to Y$
    - Either $\mathrm{id}_Y : Y \to Y$
    - Or $g \circ \mathrm{untag}_T : \mathbb{O} \to T \to Y$
  - $\mathrm{try}\{f\}\mathrm{catch}(T \Rightarrow g) = \downarrow(\mathrm{catch}(T \Rightarrow g) \circ f)$

⇨ is a propagator: try bounds the scope of catch

Control flow for $\texttt{try}\{\texttt{f}\}\texttt{catch}(\texttt{T}_1 \Rightarrow \texttt{g}_1 \mid \texttt{T}_2 \Rightarrow \texttt{g}_2 \ldots)$



Soundness of the inference system

- **Theorem**: the associated proof system is **sound**
- **Proof**:
  - from the key properties we have proofs for:

$$\frac{T \in Exc}{untag_T \circ tag_T \sim id_T}$$

$$\frac{R, T \in Exc \quad R \neq T}{untag_T \circ tag_R \sim []_T \circ tag_R}$$

1. Propagator propagates $g^{(1)} \circ []_X \equiv []_y$
2. Annihilation untag-tag $tag_T \circ untag_T \equiv id_L$
3. Annihilation catch-raise $try\{f\}catch(T \Rightarrow throw_{T,Y}) \equiv f$
4. Commutation untag-untag
   $(untag_T + id_s) \circ untag_S \equiv (id_T + untag_s) \circ untag_T$
5. Commutation catch-catch
   for S and T without any common subtype
   $try\{f\}catch(T \Rightarrow g \mid S \Rightarrow h) \equiv try\{f\}catch(S \Rightarrow h \mid T \Rightarrow g)$

...

Completeness of the inference system

- With respect to the decorated logic for exceptions
  - A theory $\mathcal{T}$ is **consistent** if there is an equation $\notin \mathcal{T}$
  - $\mathcal{T}'$ **pure extension**: generated by $\mathcal{T}$ + equations of pure terms
  - $\mathcal{T}'$ **proper extension**: extension but not pure
  - $\mathcal{T}$ Hilbert-Post **complete**:
    - consistent + no consistent proper extension

- **Theorem**

  The core language for exceptions is **Hilbert-Post complete**

- **Proof**

  every equation between catchers or propagators is equivalent to several equations between pure terms …

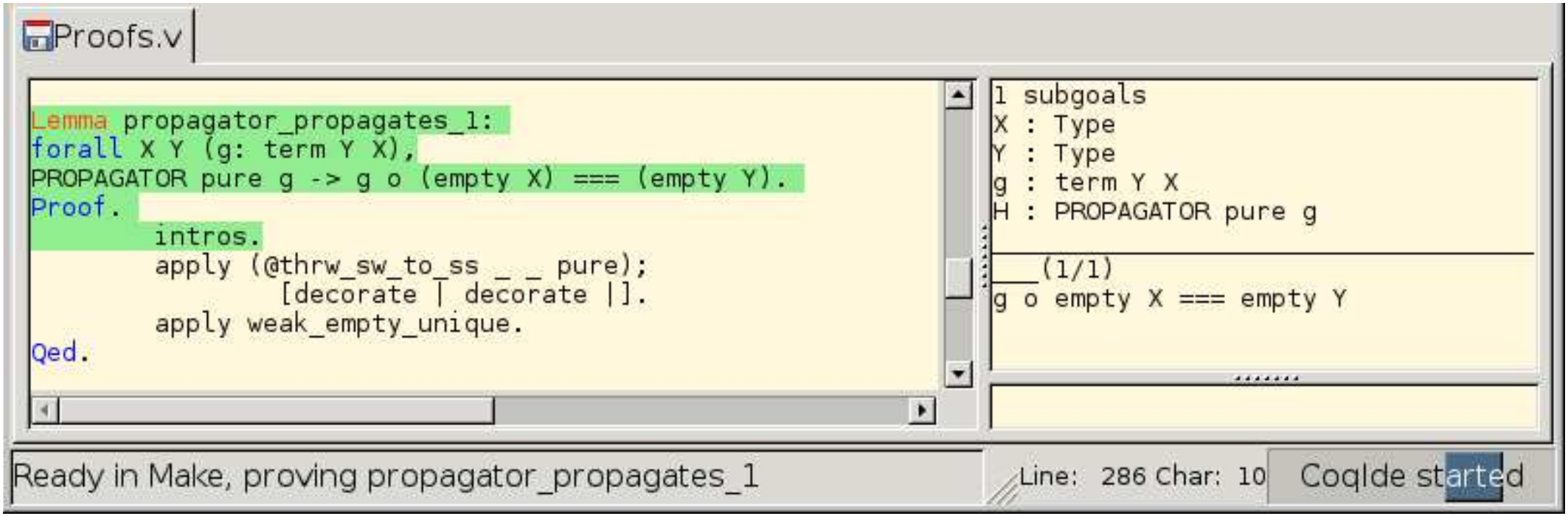
Coq

- Formalization in Coq of the exception effect
  - Dependent types for terms, decorations, equations …
  - Allows to naturally handle composition of terms

- Proving Hilbert-Post completeness in Coq
  - 4 pages of mathematical proof (LNCS style)
  - 8 pages of proof in Coq
    - 👍 One particular case forgotten
    - 👍 One mistake found in a preliminary version
    - • 1306 tactics
    - ☺ 4 seconds to formally check the proof of the theorem

ex: a propagator $g^{(1)}$ propagates exceptions

$$\dfrac{\dfrac{\dfrac{\dfrac{X}{[\,]_X : \mathbb{O} \to X} \quad g : X \to Y}{g \circ [\,]_X : \mathbb{O} \to Y}}{g \circ [\,]_X \sim [\,]_Y} \quad \dfrac{g^{(1)} \quad \dfrac{\dfrac{X}{[\,]_X^{(0)}}}{[\,]_X^{(1)}}}{(g \circ [\,]_X)^{(1)}} \quad \dfrac{\dfrac{Y}{[\,]_Y^{(0)}}}{[\,]_Y^{(1)}}}{g \circ [\,]_X \equiv [\,]_Y}$$



Towards proving modular rank algorithm

- First test
  - By hand
  - 2 x 2 explicit matrix
  - 14 variables
  - 1 exception
  - 2 loops (ext. gcd)
  - 5 if-then-else

⇨ 47 SLOC

⇨ 1215 tactics

☹ 20 minutes

IMP_Proofs.v

```
Lemma IMP_CICM: forall (a b c d m t r u u1 q g g1: Loc), forall (e: EName),
{{  a ::= (const 2) ;; (b ::= (const 1)) ;; (c ::= (const 3)) ;;
   (d ::= (const 4)) ;; (m ::= (const 6)) ;;
   IFB (loc a ?== const 0)
     THEN (t ::= (loc a) ;;  (a ::= (loc b)) ;;  (b ::= (loc t)) ;;
          (t ::= (loc c)) ;;  (c ::= (loc d)) ;;  (d ::= (loc t)))
   ELSE (SKIP)
     ENDIF ;;

   IFB (loc a ?== const 0)
     THEN (t ::= (loc a) ;;  (a ::= (loc c)) ;;  (c ::= (loc t)) ;;
          (t ::= (loc b)) ;;  (b ::= (loc d)) ;;  (d ::= (loc t)))
     ELSE (SKIP)
   ENDIF ;;

   IFB (loc a ?== const 0)
     THEN (IFB (loc b ?== const 0)
             THEN (r ::= (const 0))
             ELSE (r ::= (const 1))
           ENDIF)
     ELSE(TRY (u ::= (const 0)) ;;  (u1 ::= (const 1)) ;; (g1 ::= (loc a)) ;;  (g ::= (loc m)) ;;

              (WHILE (loc g1) >> (const 0)
                DO(q ::= (loc g /// loc g1) ;;
                  (t ::= (loc u --- loc q *** loc u1)) ;; (u ::= (loc u1)) ;; (u1 ::= (loc t)) ;;
                  (t ::= (loc g --- loc q *** loc g1)) ;; (g ::= (loc g1)) ;; (g1 ::= (loc t)))
              ENDWHILE) ;;

              IFB (loc g ?!== const 1)
               THEN (THROW e)
              ELSE (SKIP)
              ENDIF
           CATCH e => (m ::= (loc m /// loc g) ;;
                      (u ::= (const 0)) ;;  (u1 ::= (const 1)) ;; (g1 ::= (loc a)) ;;  (g ::= (loc m)) ;;
                      (WHILE (loc g1) >> (const 0)
                        DO(q ::= (loc g /// loc g1) ;;
                          (t ::= (loc u --- loc q *** loc u1)) ;; (u ::= (loc u1)) ;; (u1 ::= (loc t)) ;;
                          (t ::= (loc g --- loc q *** loc g1)) ;; (g ::= (loc g1)) ;; (g1 ::= (loc t)))
                      ENDWHILE)) ;;

           (d ::= (loc d --- loc c *** (loc u *** loc b) ?% (loc m))) ;;

           IFB (loc d ?== (const 0))
             THEN (r ::= (const 1))
             ELSE (r ::= (const 2))
          ENDIF)
     ENDIF}}
   ===
  {{a  ::= (const 2)  ;; (u1 ::= (const (3))) ;; (q ::= (const 2))  ;; (g ::= (const 1)) ;;
    (t ::= (const 0)) ;; (g1 ::= (const 0))  ;; (c  ::= (const 3)) ;;  (u ::= (const (-1))) ;;
    (b ::= (const 1)) ;; (m  ::= (const 3))  ;;  (d ::= (const 1))  ;; (r ::= (const 2))}}.
Proof. intros. simpl. unfold TRY_CATCH. apply (@thrw_sw_to_ss _ _ rw); [edecorate | edecorate |].
 (*setoid_rewrite <- weak_downcast.*) (*apply ss_to_sw.*) rewrite(@IMP_ex1 a b 2 1); [simpl | apply locdif].
 do 4 setoid_rewrite assoc at 1. setoid_rewrite <- assoc at 2. rewrite(@IMP_ex1 a c 2 3);
   [simpl | apply locdif]. setoid_rewrite assoc at 2.
 setoid_rewrite <- assoc at 3. rewrite(@IMP_ex1 a d 2 4); [simpl | apply locdif]. setoid_rewrite assoc at 3.
 setoid_rewrite <- assoc at 4. rewrite(@IMP_ex1 a m 2 6); [simpl | apply locdif]. setoid_rewrite assoc at 4.
```

Ready in Make, proving IMP_CICM

Formalized so far in Coq

- Terms: $X \rightarrow Y$
- Imperative syntax IMP:
  - if-then-else ; while loops ;
  - arithmetic (integers) expressions; Boolean expressions
  - Variable assignment
- Combined Decorations
  - Memory effect (lookup / update variables)
  - One exception effect (throw / catch)

Work in progress

- In the decorated framework
  - Higher-order logic (functional programming)
  - Composition of effects
- Parser from C to Coq-like subset of C constructs
- In Coq: several orders of magnitude to gain in speed
  - Implicit variables
  - Functions
  - Automatic composition of effects