Verifying a plaftorm for digital imaging: a multi-tool strategy*

J. Heras[1], G. Mata[2], A. Romero[2], J. Rubio[2], and R. Sáenz[2]

[1]School of Computing, University of Dundee (UK) -
[2]Department of Mathematics and Computer Science,
University of La Rioja (Spain)

Calculemus 2013

July 8, 2013

*Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Commission FP7, STREP project ForMath, n. 243847

Motivation

ForMath project: Formalisation of mathematics.

Motivation

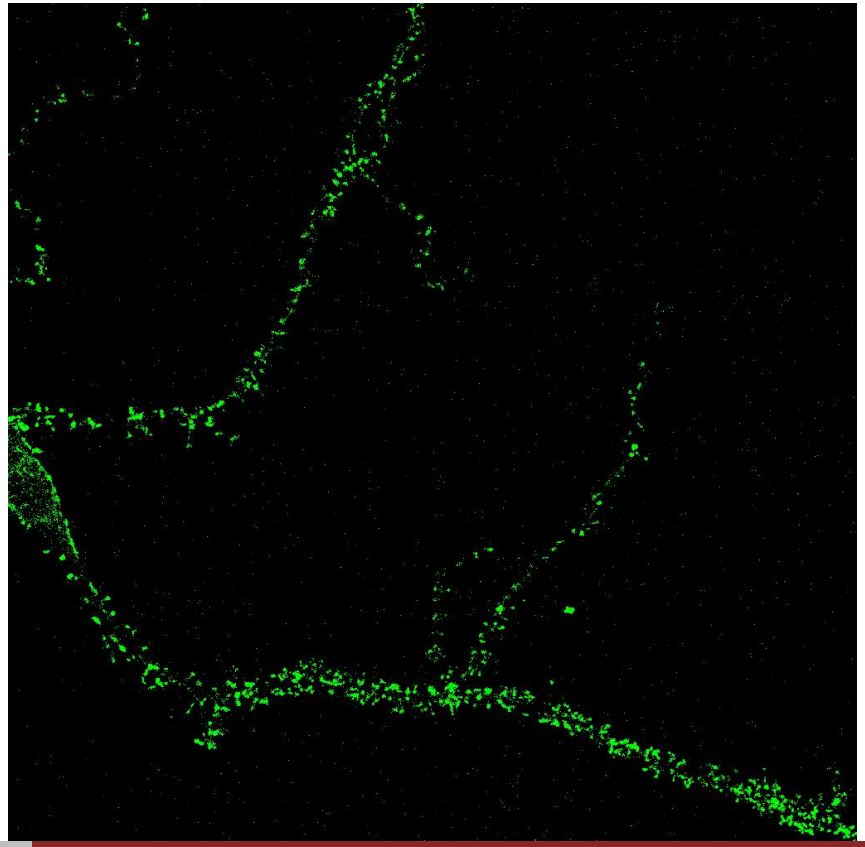
ForMath project: Formalisation of mathematics.
One of the task of La Rioja node: Application of formal algebraic topology libraries to biomedical imaging (reliable software).

Motivation

ForMath project: Formalisation of mathematics.
One of the task of La Rioja node: Application of formal algebraic topology libraries to biomedical imaging (reliable software).
Images provided by Miguel Morales' biomedical team.

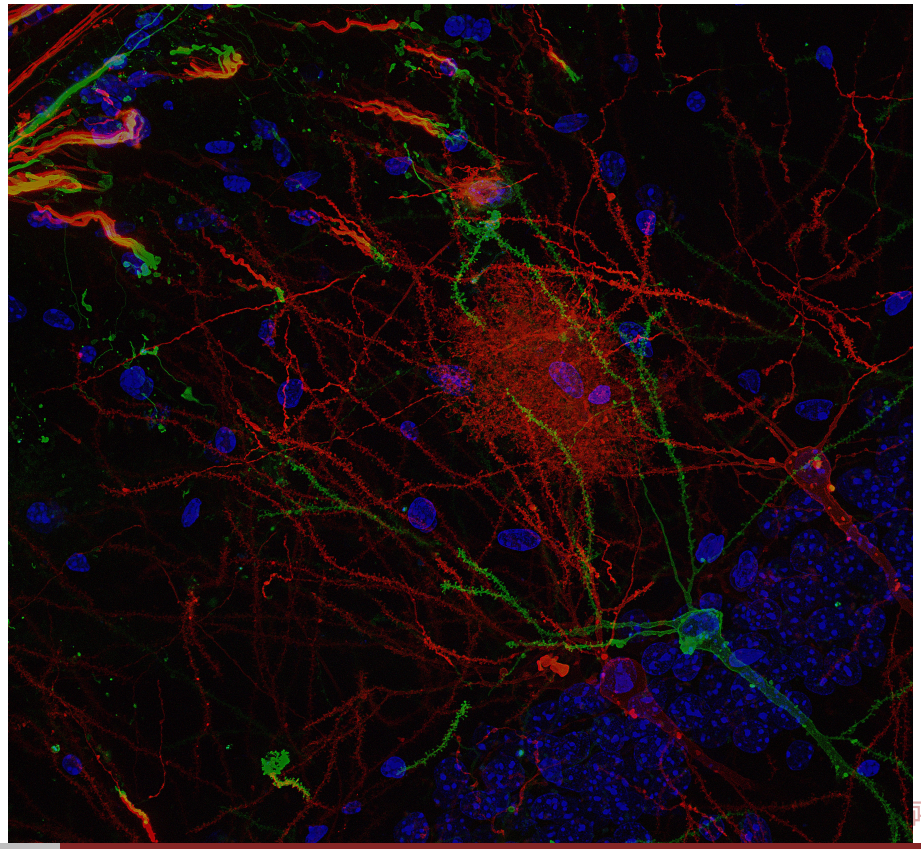
Motivation

ForMath project: Formalisation of mathematics.
One of the task of La Rioja node: Application of formal algebraic topology libraries to biomedical imaging (reliable software).
Images provided by Miguel Morales' biomedical team.

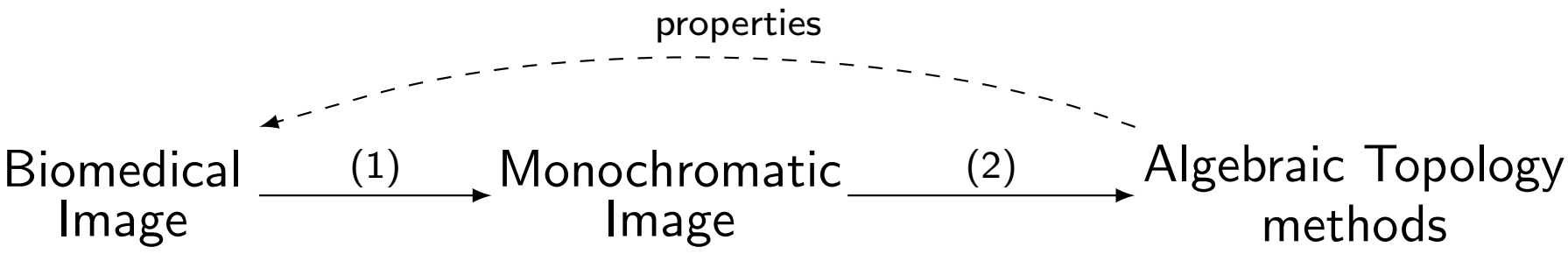


Motivation

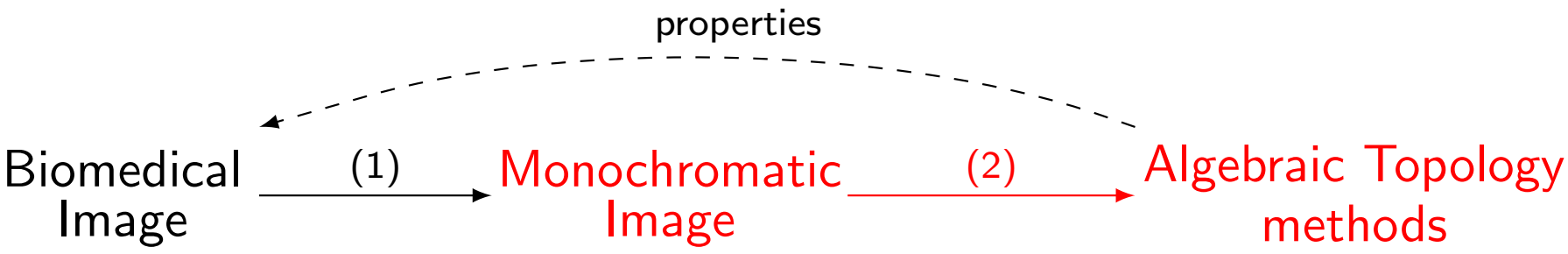
ForMath project: Formalisation of mathematics.
One of the task of La Rioja node: Application of formal algebraic topology libraries to biomedical imaging (reliable software).
Images provided by Miguel Morales' biomedical team.



Our approach



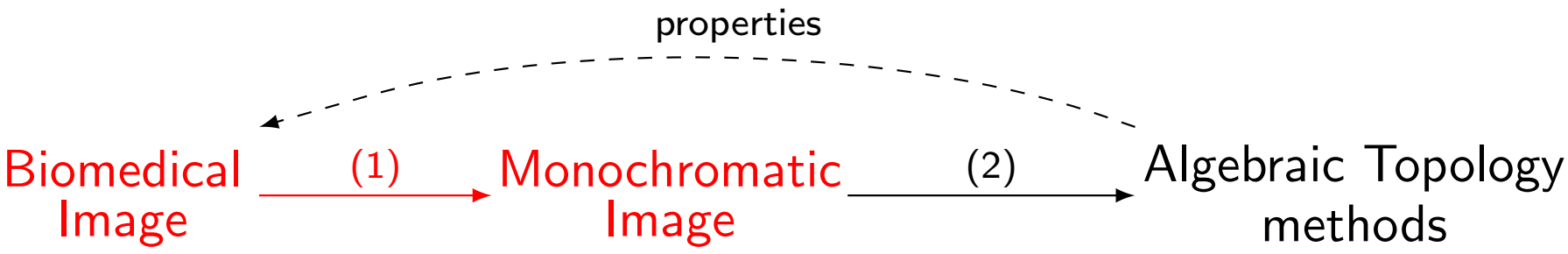
Our approach



Up to now: formal verification of Step (2).

J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza, and V. Siles. Towards a certified computation of homology groups for digital images, Proceedings 4th International Workshoph on Computational Topology in Image Context (CTIC'12), LNCS, vol. 7309, pages 49-57, 2012.

Our approach



Up to now: formal verification of Step (2).

J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza, and V. Siles. Towards a certified computation of homology groups for digital images, Proceedings 4th International Workshoph on Computational Topology in Image Context (CTIC'12), LNCS, vol. 7309, pages 49-57, 2012.

In this talk: towards a formal verification of Step (1).

Table of Contents

1. Tools and method
2. Verification of actual imaging code
3. Conclusions and Further work

Table of Contents

1. Tools and method
2. Verification of actual imaging code
3. Conclusions and Further work

Fiji/ImageJ

Fiji/ImageJ:

- Java programs used in life-science and biomedicine.
- Process and analyse biomedical images.
- Extensible by means of plugins.
- Functionality: binarise images, application of filters, maximum projection of stack of images, ...

Fiji/ImageJ

Fiji/ImageJ:

- Java programs used in life-science and biomedicine.
- Process and analyse biomedical images.
- Extensible by means of plugins.
- Functionality: binarise images, application of filters, maximum projection of stack of images, ...

Our plugins:

- SynapCountJ: count number of synapses.
- NeuronPersistentJ: detect neuronal structure.

Why/Krakatoa

Why/Krakatoa:

- Environment for proving the correctness of Java programs annotated with JML.
- Three components:
  - Krakatoa: annotated Java program $\rightarrow$ Why input.
  - Why tool: compute proof obligations.
  - Automated provers: discharge proof obligations.

Why/Krakatoa

Why/Krakatoa:

- Environment for proving the correctness of Java programs annotated with JML.
- Three components:
  - Krakatoa: annotated Java program $\rightarrow$ Why input.
  - Why tool: compute proof obligations.
  - Automated provers: discharge proof obligations.

. . . but some proof obligations cannot be automatically discharge.

Why/Krakatoa

Why/Krakatoa:

- Environment for proving the correctness of Java programs annotated with JML.
- Three components:
  - Krakatoa: annotated Java program $\rightarrow$ Why input.
  - Why tool: compute proof obligations.
  - Automated provers: discharge proof obligations.

. . . but some proof obligations cannot be automatically discharge. In those cases, Why generates Coq (and other proof assistants) code.

Coq – ACL2

Coq:

- Interactive proof assistant.
- Constructive higher-order logic based on the CoC.
- Formalisation of relevant mathematical results.

Coq – ACL2

Coq:

- Interactive proof assistant.
- Constructive higher-order logic based on the CoC.
- Formalisation of relevant mathematical results.

ACL2:

- A Computational Logic for Applicative Common Lisp.
- Automated-Interactive prover.
- Applied in formal methods projects of industrial and commercial interest.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code $\rightarrow$ compilable Krakatoa code.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code $\rightarrow$ compilable Krakatoa code.
2. Specify Java programs.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code $\rightarrow$ compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code $\rightarrow$ compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code → compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?

   Yes The verification has ended.

   No Goto 5.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code → compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?
   - Yes The verification has ended.
   - No Goto 5.
5. Are the failed attempts under-specified?

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code → compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?
   - Yes The verification has ended.
   - No Goto 5.
5. Are the failed attempts under-specified?
   - Yes Goto 2.
   - No Goto 6.
6. Extract Coq code.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code → compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?
   - Yes The verification has ended.
   - No Goto 5.
5. Are the failed attempts under-specified?
   - Yes Goto 2.
   - No Goto 6.
6. Extract Coq code.
7. Coq code → ACL2 code.

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code → compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?
   - Yes The verification has ended.
   - No Goto 5.
5. Are the failed attempts under-specified?
   - Yes Goto 2.
   - No Goto 6.
6. Extract Coq code.
7. Coq code → ACL2 code.
8. Are all the statements automatically proved in ACL2?

The method

Our method to verify the correctness of Fiji/ImageJ code:

1. Fiji code → compilable Krakatoa code.
2. Specify Java programs.
3. Apply the Why tool.
4. Proof obligations are discharged?
   - Yes The verification has ended.
   - No Goto 5.
5. Are the failed attempts under-specified?
   - Yes Goto 2.
   - No Goto 6.
6. Extract Coq code.
7. Coq code → ACL2 code.
8. Are all the statements automatically proved in ACL2?
   - Yes The verification has ended.
   - No Are other specifications needed? Yes, Goto 2. No . . .

The method

Our method to verify the correctness of Fiji/ImageJ code:

Table of Contents

1. Tools and method
2. Verification of actual imaging code
   - Challenges to formalise Fiji code
   - Transforming Fiji/Java to Krakatoa/Java
   - Specifying programs for digital imaging
   - The role of ACL2
   - Demo
3. Conclusions and Further work

Challenges to formalise Fiji code

Challenges:

- Characteristics of Fiji code: developed by several authors, messy, contains redundancies, dead code, ...
- Fiji is a big software system not devised to be formalised.

Challenges to formalise Fiji code

Challenges:

- Characteristics of Fiji code: developed by several authors, messy, contains redundancies, dead code, ...
- Fiji is a big software system not devised to be formalised.
- Why/Krakatoa does not support the complete Java programming language.
- Specification of programs.

Challenges to formalise Fiji code

Challenges:

- Characteristics of Fiji code: developed by several authors, messy, contains redundancies, dead code, ...
- Fiji is a big software system not devised to be formalised.
- Why/Krakatoa does not support the complete Java programming language.
- Specification of programs.
- ATPs does not automatically discharge all proof obligations.
- ITPs lack automation.

Transforming Fiji-Java to Krakatoa-Java

Making Fiji/Java programs compilable in Krakatoa:

1. Delete annotations.
2. Move the classes that are referenced into Why folder.
3. Reproduce the behaviour of the class that we want to compile.
4. Remove import clauses.
5. Remove package declarations.
6. Rebuild native methods.
7. Add a clause in if-else structures in order to remove "Uncaught exceptions".
8. Remove debugging useless references.
9. Modify the declaration of some variables to avoid syntax errors.
10. Change the way that Maximum and Minimum float numbers are written.

Transforming Fiji-Java to Krakatoa-Java

1. Delete annotations.

JML annotations are placed between `\*@ and @*\`.
Remove Java annotations preceded by `@`.

Transforming Fiji-Java to Krakatoa-Java

1. Delete annotations.

JML annotations are placed between `\*@ and @*\`.
Remove Java annotations preceded by `@`.

6. Rebuild native methods.

Java allows the use of native methods written in C/C++ (dependent on hardware/OS).
Methods in the class *Math*: exponential, logarithm, square root, trigonometric functions, ...
Native methods cannot be verified in Krakatoa $\Longrightarrow$ Rewrite them.

Transforming Fiji-Java to Krakatoa-Java

1. Delete annotations.

JML annotations are placed between `\*@ and @*\`.
Remove Java annotations preceded by `@`.

6. Rebuild native methods.

Java allows the use of native methods written in C/C++ (dependent on hardware/OS).
Methods in the class *Math*: exponential, logarithm, square root, trigonometric functions, ...
Native methods cannot be verified in Krakatoa $\Longrightarrow$ Rewrite them.

10. Change Maximum and Minimum float numbers format.

Used to avoid overflow errors.
They generate an error due to the eP exponent.
Conversion from `0x1.fffffeP+127d` to `3.4028235e+38f`.

Specifying programs for digital imaging

Specification in Krakatoa is not straightforward.

Specifying programs for digital imaging

Specification in Krakatoa is not straightforward.
Specification of preconditions and postconditions using JML to certify the correctness of the programs.

Specifying programs for digital imaging

Specification in Krakatoa is not straightforward.
Specification of preconditions and postconditions using JML to certify the correctness of the programs.

- Precondition: `requires`.
- Postcondition: `ensures`.
- Notations: `\result` and `old`.

Example 1: Translating an object

```
 private double x;
 private double y;

 /*@ ensures x == \old(x) + dx && y == \old(y) + dy; @*/

 public void translate(final double dx, final double dy) {
    this.x += dx; this.y += dy;
}
```

Example 1: Translating an object

```
 private double x;
 private double y;

 /*@ ensures x == \old(x) + dx && y == \old(y) + dy; @*/

 public void translate(final double dx, final double dy) {
    this.x += dx; this.y += dy;
}
```

Small demo in Krakatoa.

Example 2: x^n

```
static double power(double x, int n){
  int i=0;
  double pot=1;
  while (i<n) {
     pot=pot*x;
     i++;
     }
  return pot;
  }
```

Example 2: x^n

```
/*@ axiomatic Power {
  @ logic real lpower(real x, integer n);
  @
  @ axiom power_zero: \forall real x; lpower(x,0) == 1.0;
  @
  @ axiom power_one: \forall real x; lpower(x,1) == x;
  @
  @ axiom power_two: \forall real x; lpower(x,2) == x*x;
  @
  @ axiom power_sum:
  @ \forall real x, integer i j; lpower(x,i+j) == lpower(x,i)+lpower(x,j);
  @
  @ axiom power_sum2:
  @ \forall real x, integer i j; lpower(x,i*j) == lpower(lpower(x,i),j);
  @
  @ axiom power_multi:
  @ \forall real x y, integer i; lpower(x*y,i) == lpower(x,i) * lpower(y,i);
  @
  @ axiom power_general1:
  @    \forall real x , integer i;x > 1 && i%2 == 1
  @ ==> lpower(x,i) == lpower(lpower(x,2),i/2)*x;
  @  ... }
  @*/
```

Example 2: x^n

```
/*@ requires n >=0;
  @ ensures \result==lpower(x,n);
  @*/

  static double power(double x, int n){
     int i=0;
     double pot=1;
     /*@ loop_invariant
       @ i <= n && pot==lpower(x,i) ;
       @ loop_variant n-i ;
       @*/
     while (i<n) {
        pot=pot*x;
        i++;
        }
    /*@ assert i==n; @*/
    return pot;
 }
```

Example 2: x^n

```
/*@ requires n >=0;
  @ ensures \result==lpower(x,n);
  @*/

  static double power(double x, int n){
     int i=0;
     double pot=1;
     /*@ loop_invariant
       @ i <= n && pot==lpower(x,i) ;
       @ loop_variant n-i ;
       @*/
     while (i<n) {
        pot=pot*x;
        i++;
        }
    /*@ assert i==n; @*/
    return pot;
 }
```

Example 2: x^n

```
/*@ requires n >=0;
  @ ensures \result==lpower(x,n);
  @*/

  static double power(double x, int n){
     int i=0;
     double pot=1;
     /*@ loop_invariant
       @ i <= n && pot==lpower(x,i) ;
       @ loop_variant n-i ;
       @*/
     while (i<n) {
        pot=pot*x;
        i++;
        }
    /*@ assert i==n; @*/
    return pot;
 }
```

Example 2: x^n

```
/*@ requires n >=0;
  @ ensures \result==lpower(x,n);
  @*/

  static double power(double x, int n){
     int i=0;
     double pot=1;
     /*@ loop_invariant
       @ i <= n && pot==lpower(x,i) ;
       @ loop_variant n-i ;
       @*/
     while (i<n) {
        pot=pot*x;
        i++;
        }
    /*@ assert i==n; @*/
    return pot;
 }
```

Example 2: x^n

```
/*@ requires n >=0;
  @ ensures \result==lpower(x,n);
  @*/

  static double power(double x, int n){
     int i=0;
     double pot=1;
     /*@ loop_invariant
       @ i <= n && pot==lpower(x,i) ;
       @ loop_variant n-i ;
       @*/
     while (i<n) {
        pot=pot*x;
        i++;
        }
    /*@ assert i==n; @*/
    return pot;
 }
```

Small demo in Krakatoa.

Using ACL2 in our framework

How ACL2 can help in the proof verification process?

Using ACL2 in our framework

How ACL2 can help in the proof verification process?

- Scope of automated provers is smaller than the one of ACL2.
- Interactive provers lack automation.

Using ACL2 in our framework

How ACL2 can help in the proof verification process?

- Scope of automated provers is smaller than the one of ACL2.
- Interactive provers lack automation.

Coq2ACL2 – a Proof General extension.

Using ACL2 in our framework

How ACL2 can help in the proof verification process?

- Scope of automated provers is smaller than the one of ACL2.
- Interactive provers lack automation.

Coq2ACL2 – a Proof General extension.

- Transforms Coq statements generated by Why to ACL2.
- Automatically sends the ACL2 statements to ACL2.
- Displays the proof attempt generated by ACL2.

The I2EA framework

Original aim:

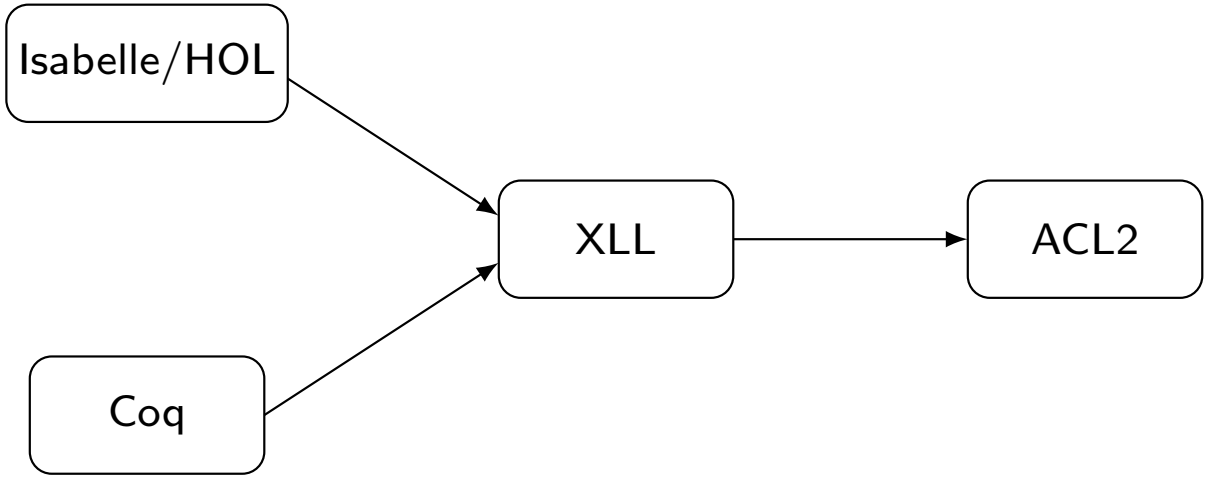
- Isabelle/HOL theories $\rightarrow$ ACL2.
- Isabelle/HOL first-order expressions $\rightarrow$ schema of the proof in ACL2.

The I2EA framework

XLL (Xmall Logical Language):

- Intermediate language to port Isabelle/HOL theories to ACL2 and Ecore (UML + OCL).
- Transformations are done by means of XSLT.

The I2EA framework



Integration of Coq in this framework.

- Reuse XLL and XSLT transformations.
- Coq statements $\rightarrow$ ACL2.
- Coq2ACL2:

$$\text{Coq} \xrightarrow{\text{Lisp program}} \text{XLL} \xrightarrow{\text{XSLT}} \text{ACL2}$$

Demo

Motivation:

- Images acquired by microscopy techniques.
- Different planes from the same sample (same XY plane different Z levels).
- Maximum projection of the stack of images.
- Maximum projection methods: maximum intensity or standard deviation.

Demo

Motivation:

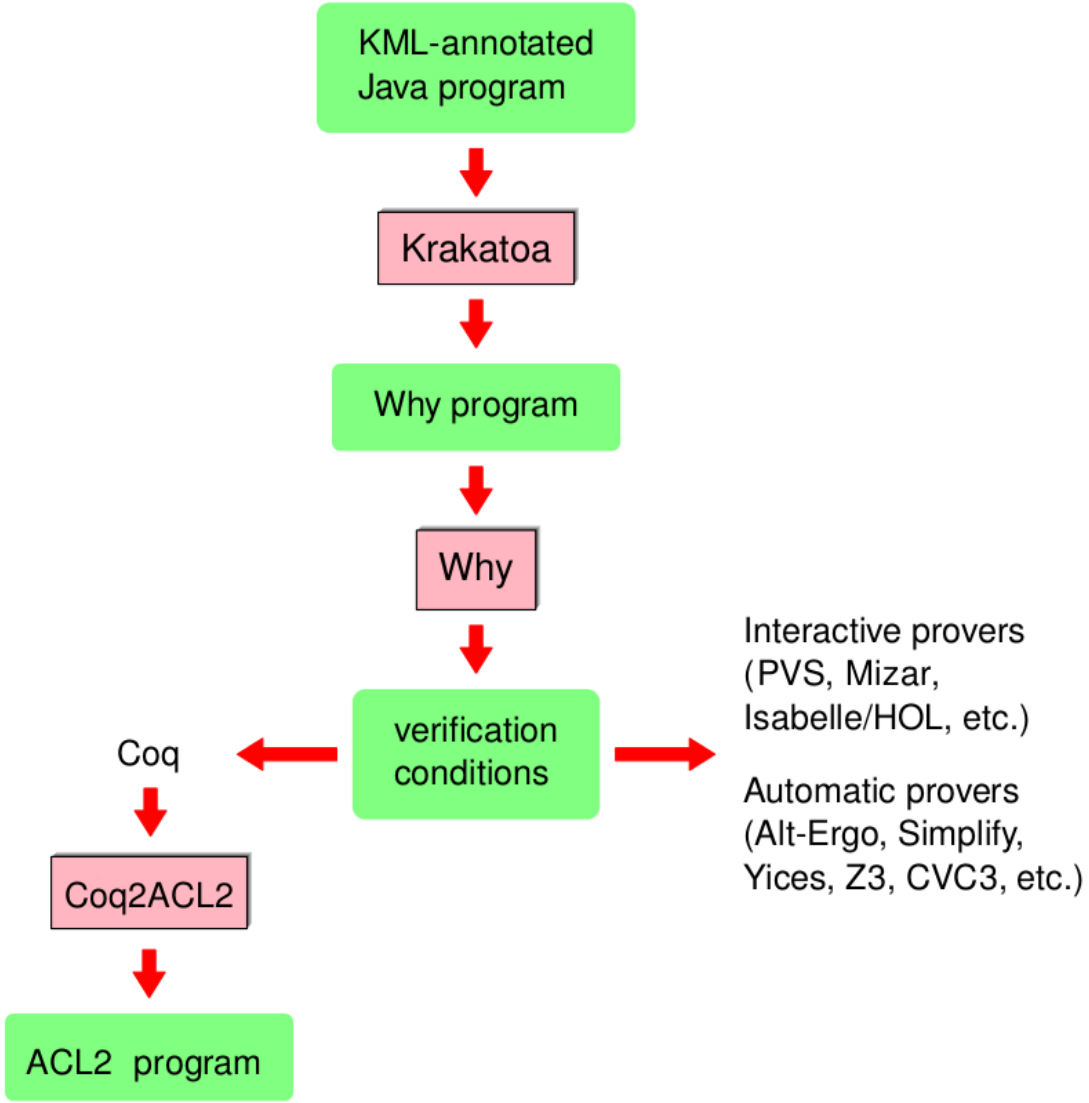
- Images acquired by microscopy techniques.
- Different planes from the same sample (same XY plane different Z levels).
- Maximum projection of the stack of images.
- Maximum projection methods: maximum intensity or standard deviation.

Demo: correctness of the calculateStdDev method.

Table of Contents

1. Tools and method
2. Verification of actual imaging code
3. Conclusions and Further work

Our approach to verify Java code



Conclusions and Further work

- Conclusions:
  - Verification of Fiji/ImageJ Java code.
  - Isolate relevant parts of programs to formalise them.
  - Fiji is unsound but the errors are minor and can be easily corrected.
  - Integration of several tools keeping transformations as simple as possible.
  - Reuse of I2EA environment to combine systems.

Conclusions and Further work

- Conclusions:
  - Verification of Fiji/ImageJ Java code.
  - Isolate relevant parts of programs to formalise them.
  - Fiji is unsound but the errors are minor and can be easily corrected.
  - Integration of several tools keeping transformations as simple as possible.
  - Reuse of I2EA environment to combine systems.
- Further work:
  - Automatic transformation from Java code to Krakatoa code.
  - Automatic discovery of proof invariants.
  - Reconstruction of ACL2 proofs in Coq.
  - Full certification of SynapCountJ and NeuronPersistentJ.

Verifying a plaftorm for digital imaging: a multi-tool strategy*

J. Heras[1], G. Mata[2], A. Romero[2], J. Rubio[2], and R. Sáenz[2]

[1]School of Computing, University of Dundee (UK) -
[2]Department of Mathematics and Computer Science,
University of La Rioja (Spain)

Calculemus 2013

July 8, 2013

*Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Commission FP7, STREP project ForMath, n. 243847