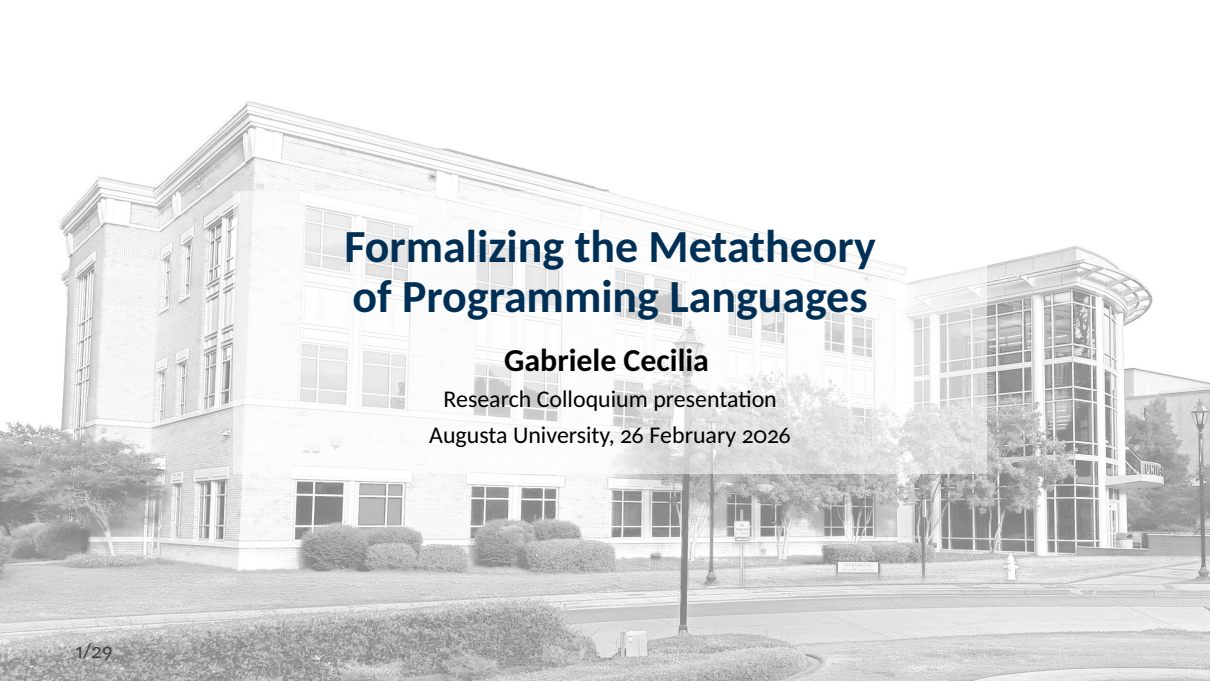


Formalizing the Metatheory of Programming Languages

Gabriele Cecilia

Research Colloquium presentation

Augusta University, 26 February 2026

A quick recap…

Formal Methods for Reversible Concurrent Calculi:

Studying and improving reversible concurrent calculi, by formalizing their definitions and properties with proof assistants

A quick recap…

Formal Methods for Reversible Concurrent Calculi:

Studying and improving reversible concurrent calculi, by formalizing their definitions and properties with proof assistants

Table of Contents

- ▶ Introduction
- ▶ Formal methods
- ▶ Formalizing programming languages
- ▶ Encoding syntax
- ▶ Encoding variables
- ▶ Conclusion

A Leading Question

Dr. Arman Adibi, Research Colloquium, 19 February:

“Are AI systems reliable?”

A Leading Question

This talk:

“Is software reliable?”

Software Bugs

Everyone uses some kind of software, in their research or everyday life

However, we can all figure annoying examples of software bugs

Software Bugs - Examples

Software Bugs - Examples

Therac 25

But not only software...

→ Software bugs are common, but they are not the only source of failure

But not only software...

→ Software bugs are common, but they are not the only source of failure

→ Software is written in programming languages

But not only software...

→ Software bugs are common, but they are not the only source of failure

→ Software is written in programming languages

→ Compilers translate code from one language to another

But not only software...

→ Software bugs are common, but they are not the only source of failure

→ Software is written in programming languages

→ Compilers translate code from one language to another

...and errors can occur *at each of these stages*

Table of Contents

- ▶ Introduction
- ▶ Formal methods
- ▶ Formalizing programming languages
- ▶ Encoding syntax
- ▶ Encoding variables
- ▶ Conclusion

Possible solutions

How to guarantee correctness of software, compilers, programming languages?

Possible solutions

How to guarantee correctness of software, compilers, programming languages?

- Good programming practices (code documentation, code reviews, . . .)

Possible solutions

How to guarantee correctness of software, compilers, programming languages?

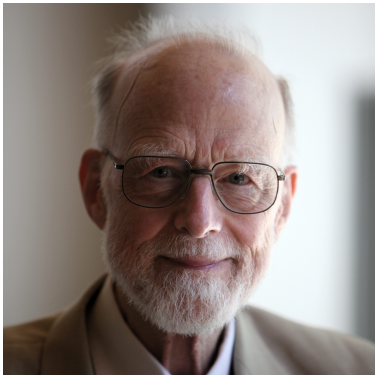
- Good programming practices (code documentation, code reviews, . . .)
- Testing
- Static analysis

Possible solutions

How to guarantee correctness of software, compilers, programming languages?

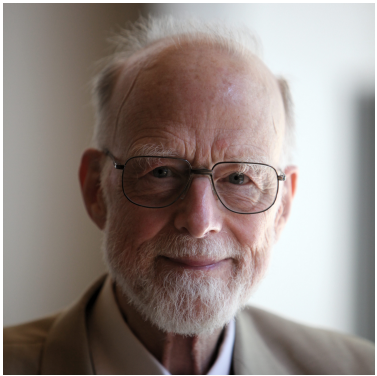
- Good programming practices (code documentation, code reviews, ...)
- Testing
- Static analysis
- Formal methods

Formal Methods



→ First developed in the 1960s (Hoare, Floyd)

Formal Methods



→ First developed in the 1960s (Hoare, Floyd)

→ Mathematics and logical principles to specify, develop and verify systems

A Taxonomy of Formal Methods

- Program logics (Hoare logic, separation logic, temporal logic ...)
- Automata (finite, Buchi, ...)
- Process algebra (CCS, π-calculus, ...)
- Model checking
- Runtime verification
- Automatic theorem proving (SAT solvers, SMT solvers, ...)
- Interactive theorem proving (Rocq, Isabelle, Agda, ...)

To sum up...

Problem: correctness of software, compilers, programming languages

To sum up...

Problem: correctness of software, compilers, programming languages

A solution: formal methods

To sum up...

Problem: correctness of software, compilers, programming languages

A solution: formal methods

A tool: proof assistants

To sum up...

Problem: correctness of software, compilers, programming languages

A solution: formal methods

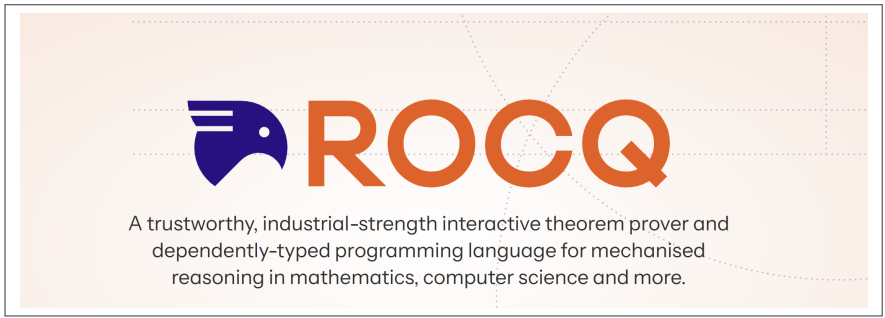
A tool: proof assistants

Goal: formalizing languages and their properties with proof assistants

Table of Contents

- ▶ Introduction
- ▶ Formal methods
- ▶ Formalizing programming languages
- ▶ Encoding syntax
- ▶ Encoding variables
- ▶ Conclusion

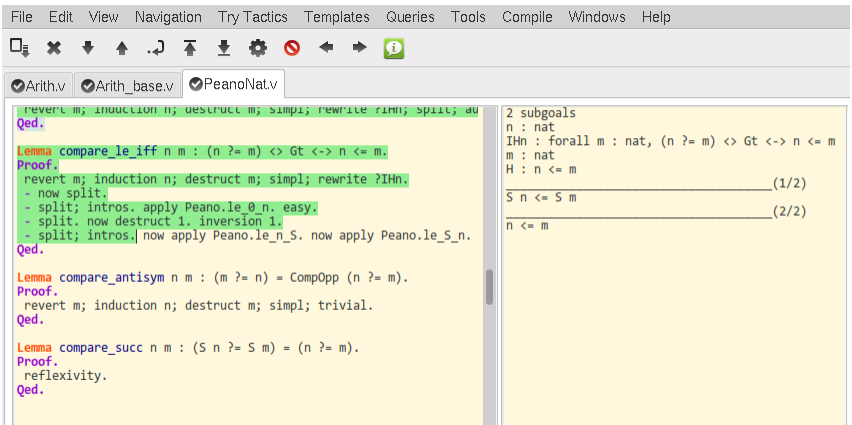
What are proof assistants?



Computer programs for constructing machine-checked mathematical proofs

Scope: programming languages, compilers, software, mathematics, …

What are their features?



- Interaction with the user (tactics, proof state, computation holes . . .)

What are their features?

- Interaction with the user (tactics, proof state, computation holes . . .)
- Automation

What are their features?

- Interaction with the user (tactics, proof state, computation holes . . .)
- Automation
- Libraries

What are their features?

- Interaction with the user (tactics, proof state, computation holes . . .)
- Automation
- Libraries
- Proofs checked by a trusted kernel

Why using them?

- Guarantee correctness of proofs

Why using them?

- Guarantee correctness of proofs
- Proof search and counterexamples search

Why using them?

- Guarantee correctness of proofs
- Proof search and counterexamples search
- Proofs with numerous cases, mostly not deep

Why using them?

- Guarantee correctness of proofs
- Proof search and counterexamples search
- Proofs with numerous cases, mostly not deep
- Increase comprehension of systems

What is the metatheory of programming languages?

Examples of theorems:

- All programs are memory safe (no illegal memory access)
- All well-typed programs eventually terminate
- Every compiled program preserves the meaning of the source program

What is the metatheory of programming languages?

Examples of theorems:

- All programs are memory safe (no illegal memory access)
- All well-typed programs eventually terminate
- Every compiled program preserves the meaning of the source program

They express properties of *all* programs in a language.

More generally, metatheorems quantify over *linguistic* structures such as programs, states, types, terms, formulas

What is needed to formalize programming languages metatheory?

Syntax	Semantics	Theorems
Programs structure	Programs behavior	Programs properties
Algebraic datatypes	Inductive definitions, recursive functions	Structural or well-founded (co)induction

What is needed to formalize programming languages metatheory?

Syntax	Semantics	Theorems
Programs structure	Programs behavior	Programs properties
Algebraic datatypes	Inductive definitions, recursive functions	Structural or well-founded (co)induction

For each, a way to represent variables and binder constructs is also needed

Table of Contents

- ▶ Introduction
- ▶ Formal methods
- ▶ Formalizing programming languages
- ▶ Encoding syntax
- ▶ Encoding variables
- ▶ Conclusion

A toy example

Consider a simple language of arithmetic expressions:

- Natural numbers: $0, 1, 2, \ldots$

A toy example

Consider a simple language of arithmetic expressions:

- Natural numbers: $0, 1, 2, \ldots$
- Arithmetic expressions:

  $A_1, A_2 ::= n \mid A_1 + A_2 \mid A_1 - A_2 \mid A_1 \times A_2$

  Example:
  $A := 2 \times (5 - 3)$

How to encode natural numbers?

Peano axioms: zero and a successor (+ properties)

How to encode natural numbers?

Peano axioms: zero and a successor (+ properties)

$\rightarrow$ A type with two constructors:

```
LF nat: type =
  | zero: nat
  | succ: nat → nat
;
```

Examples:

`zero`, `succ zero`, `succ (succ zero)`,...

How to encode arithmetic expressions?

```
LF aexp: type =
   | aNum: nat → aexp                    n
   | aPlus: aexp → aexp → aexp           A1 + A2
   | aMinus: aexp → aexp → aexp          A1 − A2
   | aMult: aexp → aexp → aexp           A1 × A2
;
```

Examples:

$3 \leftrightarrow$ `aNum (succ (succ (succ zero)))`,

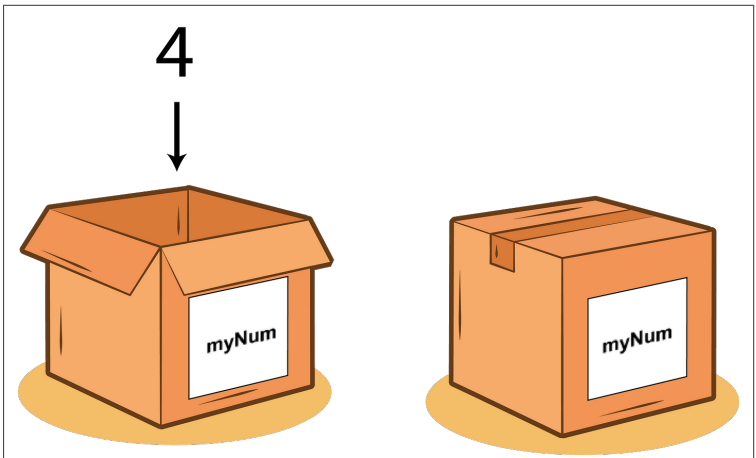
$1 - 0 \leftrightarrow$ `aMinus (aNum (succ zero)) (aNum zero)`

Notation can be simplified

Table of Contents

- ▶ Introduction
- ▶ Formal methods
- ▶ Formalizing programming languages
- ▶ Encoding syntax
- ▶ Encoding variables
- ▶ Conclusion

What is a variable?



What is a binding construct?

Example:

```
let X = 5 in (X + 1)
```

What is a binding construct?

Example:

```
let X = 5 in (X + 1)
```

$\rightarrow$ A piece of syntax introducing a variable and defining its scope

What is a binding construct?

Example:

```
let X = 5 in (X + 1)
```

$\rightarrow$ A piece of syntax introducing a variable and defining its scope

Consider now the following:

```
let X = 5 in (X + Y)
```

`X` is bound in `(X + Y)`, `Y` is free in `(X + Y)`

Renaming and substitutions

Consider the following expressions:

```
let X = 3 in (X - 2)
let Y = 3 in (Y - 2)
```

Renaming and substitutions

Consider the following expressions:

```
let X = 3 in (X - 2)
let Y = 3 in (Y - 2)
```

α-*renaming*: expressions differing only for the choice of bound names are the same

Renaming and substitutions

Consider now the following:

```
let X = 5 in
  (let Y = 2 in (X + 1))
```

↓

```
let Y = 2 in (5 + 1)
```

↓

```
5 + 1
```

```
let X = Y in
  (let Y = 2 in (X + 1))
```

↓

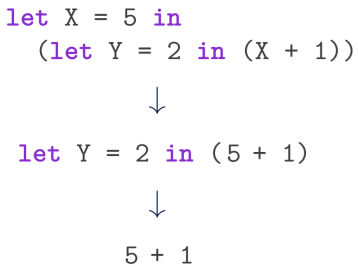
```
let Y = 2 in (Y + 1)
```

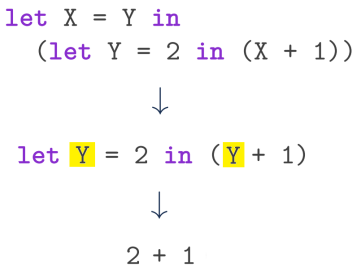
↓

```
2 + 1
```

Renaming and substitutions

Consider now the following:





Capture-avoiding substitutions: substitutions must avoid variable captures

Named Terms

Let's add variables and let expressions to the toy language seen before:

$$A_1, A_2 ::= n \mid x \mid A_1 + A_2 \mid A_1 - A_2 \mid A_1 \times A_2 \mid \text{let } x = A_1 \text{ in } A_2$$

Named Terms

Named Terms: represent variables as strings or integers

```
LF aexp: type =
   | aNum: nat → aexp
   | aPlus: aexp → aexp → aexp
   | aMinus: aexp → aexp → aexp
   | aMult: aexp → aexp → aexp
   | aVar: string → aexp                          x
   | aLet: string → aexp → aexp → aexp            let x = A1 in A2
;
```

Example: let $X = 1$ in $(X - 0)$

→ `aLet 'X' (aNum (succ zero)) (aMinus (aVar 'X') (aNum zero))`

Named Terms

Named Terms: represent variables as strings or integers

```
LF aexp: type =
   | aNum: nat → aexp
   | aPlus: aexp → aexp → aexp
   | aMinus: aexp → aexp → aexp
   | aMult: aexp → aexp → aexp
   | aVar: string → aexp                          x
   | aLet: string → aexp → aexp → aexp            let x = A1 in A2
;
```

Advantages: closer to human representation
Disadvantages: α-renaming and substitutions

De Bruijn Indices

De Bruijn Indices: variables as numbers, counting the number of binders between a variable occurrence and its corresponding binder

Example: let $X = A_1$ in (let $Y = A_2$ in $(X + Y)$)

$\rightarrow$ let ~~$X =$~~ A_1 in (let ~~$Y =$~~ A_2 in $(x_1 + x_0)$)

De Bruijn Indices

```
LF aexp: type =
  ...
  | aVar: nat → aexp
  | aLet: aexp → aexp → aexp
;
```

Example: let $X = 1$ in $(X - 0)$

→ `aLet (aNum (succ zero)) (aMinus (aVar zero) (aNum zero))`

De Bruijn Indices

```
LF aexp: type =
  ...
  | aVar: nat → aexp
  | aLet: aexp → aexp → aexp
;
```

Example: let $X = 1$ in $(X - 0)$
→ `aLet (aNum (succ zero)) (aMinus (aVar zero) (aNum zero))`

Advantages: unique representation (no α-renaming)
Disadvantages: less readable; substitutions require shifting indices; variables may have different representations

Higher-Order Abstract Syntax

HOAS: variables as arguments of meta-level functions

It uses the meta-language's own notion of variables and functions to represent object-language binding

Higher-Order Abstract Syntax

```
LF aexp: type =
  ...
  | aLet: aexp → (aexp → aexp) → aexp
;
```

Example: let $X = 1$ in $(X - 0)$
→ `aLet (aNum (succ zero)) \x.(aMinus x (aNum zero))`,
where `\x.(f x)` denotes the meta-level function mapping x to $f(x)$

Higher-Order Abstract Syntax

```
LF aexp: type =
  ...
  | aLet: aexp → (aexp → aexp) → aexp
;
```

Example: let $X = 1$ in $(X - 0)$
→ `aLet (aNum (succ zero)) \x.(aMinus x (aNum zero))`,
where `\x.(f x)` denotes the meta-level function mapping x to $f(x)$

Advantages: α-renaming and substitutions for free
Disadvantages: complicated to handle in some domains

Table of Contents

- ▶ Introduction
- ▶ Formal methods
- ▶ Formalizing programming languages
- ▶ Encoding syntax
- ▶ Encoding variables
- ▶ Conclusion

Summary

- Formal methods are techniques to ensure the correctness of software and programming languages

Summary

- Formal methods are techniques to ensure the correctness of software and programming languages
- In particular, proof assistants can be applied to verify properties of programming languages

Summary

- Formal methods are techniques to ensure the correctness of software and programming languages
- In particular, proof assistants can be applied to verify properties of programming languages
- Formalizing the metatheory of programming languages is not straightforward and requires, e.g., dealing with binding constructs

Summary

- Formal methods are techniques to ensure the correctness of software and programming languages
- In particular, proof assistants can be applied to verify properties of programming languages
- Formalizing the metatheory of programming languages is not straightforward and requires, e.g., dealing with binding constructs
- There is not a “best” binding encoding technique for all applications

Application to my current research

Reversible concurrent calculi like CCSK are particular kinds of languages with binders

$X, Y ::=$	$\mathbf{0}$	(Inactive)	$\mid \alpha.X$	(Prefix)
	$\mid \alpha[k].X$	(Keyed prefix)	$\mid X + Y$	(Sum)
	$\mid X \mid Y$	(Parallel composition)	$\mid X \setminus a$	(Restriction)

Currently, I am formalizing them in Beluga with the HOAS technique

References

- `https://ethicsunwrapped.utexas.edu/case-study/therac-25`
- Doron A. Peled. *Formal Methods*. In Sungdeok Cha, Richard N. Taylor and Kyochul Kang, editors, Handbook of Software Engineering. Springer, Cham, 2019. doi:10.1007/978-3-030-00262-6_5
- Dale Miller. *Mechanized Metatheory Revisited*. J Autom Reasoning 63, 625–665, 2019. doi:10.1007/s10817-018-9483-3
- Robbert Krebbers, Alberto Momigliano, and Brigitte Pientka. *Formalization of Programming Languages*. In Jasmin Blanchette and Assia Mahboubi, editors, Handbook of Proof Assistants. Springer, 2025. Forthcoming
- Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 1997. In preparation. Draft from April 1997 available electronically.

Thank you for listening!
Any questions?