Computational Flag Algebras in Lean

Flags in the Mountains | CSU Mountain Campus

Christoph Spiegel | Zuse Institute Berlin

2 June 2026

Two Caveats Up Front

1. **A talk about verifying results, not about obtaining them.**

   Not about a specific extremal combinatorial problem and not how one *solves* flag algebra problems. It is about how we can trust a flag algebra proof once we have one.

Two Caveats Up Front

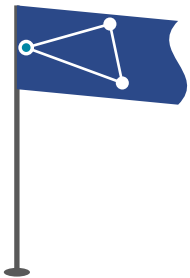
1. **A talk about verifying results, not about obtaining them.**

Not about a specific extremal combinatorial problem and not how one *solves* flag algebra problems. It is about how we can trust a flag algebra proof once we have one.

---

2. **The word ‘type’ is overloaded.**
   - In **Flag Algebras**: the fixed, labeled graph core shared by the family of all flags of that ‘type’.
   - In **Lean**: what kind of value a piece of data holds and the foundational notion of the logic itself.

To keep them apart, we call the labeled core of a flag its pole.



Flag Algebras, Certificates, and Trust

Our Running Example: Goodman's Bound

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

Our Running Example: Goodman's Bound

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

What do Flag Algebras actually give us?

1. **Notation.** Drop the redundant '$d(\cdot, H_n)$' and 'for every sequence of graphs (H_n)' throughout; just write $K_3 + I_3 \geq 1/4$ — *a real vector space spanned by graphs*.

Our Running Example: Goodman's Bound

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

What do Flag Algebras actually give us?

1. **Notation.** Drop the redundant '$d(\cdot, H_n)$' and 'for every sequence of graphs (H_n)' throughout; just write $K_3 + I_3 \geq 1/4$ — *a real vector space spanned by graphs*.
2. **A product.** Turn it into an *algebra*; in particular, every square is non-negative.

Our Running Example: Goodman's Bound

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

What do Flag Algebras actually give us?

1. **Notation.** Drop the redundant '$d(\cdot, H_n)$' and 'for every sequence of graphs (H_n)' throughout; just write $K_3 + I_3 \geq 1/4$ — *a real vector space spanned by graphs*.
2. **A product.** Turn it into an *algebra*; in particular, every square is non-negative.
3. **Chain rule.** Re-express densities at higher order, collapsing parts of the algebra.

Our Running Example: Goodman's Bound

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

What do Flag Algebras actually give us?

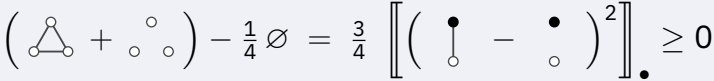
1. **Notation.** Drop the redundant '$d(\cdot, H_n)$' and 'for every sequence of graphs (H_n)' throughout; just write $K_3 + I_3 \geq 1/4$ — *a real vector space spanned by graphs*.
2. **A product.** Turn it into an *algebra*; in particular, every square is non-negative.
3. **Chain rule.** Re-express densities at higher order, collapsing parts of the algebra.
4. **Poles.** Each way of fixing finitely many labeled vertices yields its own algebra with the pole as unit, related to the others by order-preserving maps.

A Flag Algebra Certificate

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

Combining all four, someone can hand you a certificate for this statement:



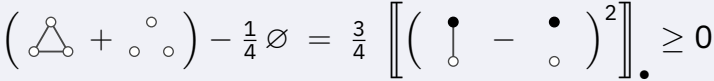
the filled vertex marks the labeled pole.

A Flag Algebra Certificate

Goodman (1959)

$\liminf_{n\to\infty} d(K_3, H_n) + d(I_3, H_n) \geq 1/4$ for every sequence of graphs $(H_n)_{n\in\mathbb{N}}$.

Combining all four, someone can hand you a certificate for this statement:

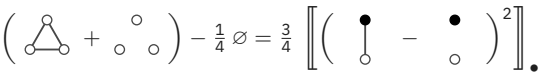


the filled vertex marks the labeled pole.

Crucially, finding the certificate should be the hard part. Checking a certificate should not: it has to be a purely mechanical computation that runs on modest hardware.*

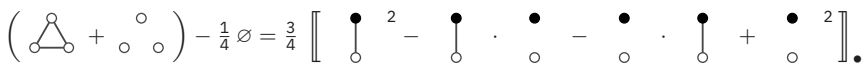
* ideally

A Mechanical Verification of a Flag Algebra Certificate



The claimed certificate

A Mechanical Verification of a Flag Algebra Certificate

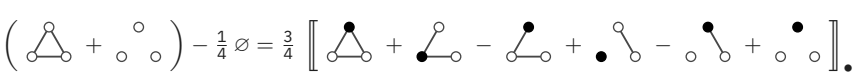


Apply linearity of the product

A Mechanical Verification of a Flag Algebra Certificate

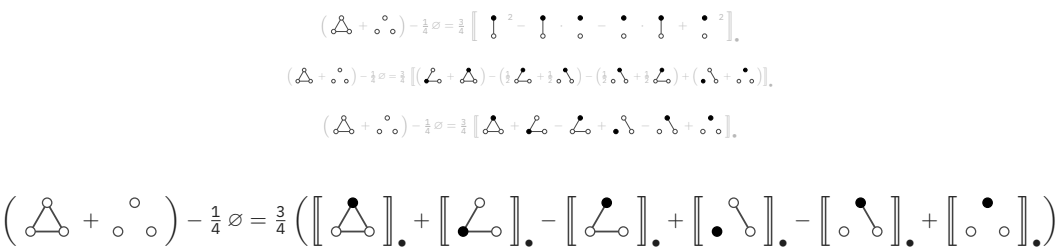
Resolve all products (and canonizing graphs)

A Mechanical Verification of a Flag Algebra Certificate



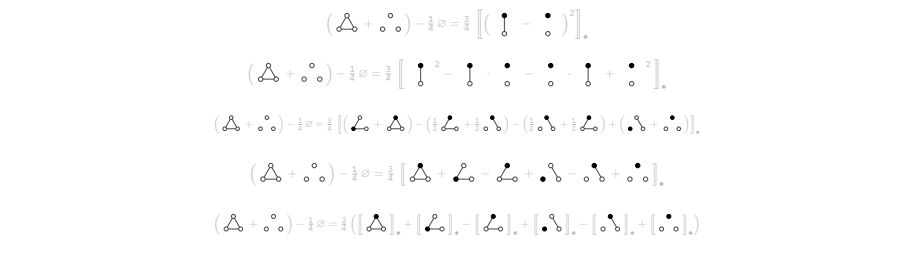
Canonize through commutativity and distributivity

A Mechanical Verification of a Flag Algebra Certificate



Apply linearity of the downward operator

A Mechanical Verification of a Flag Algebra Certificate

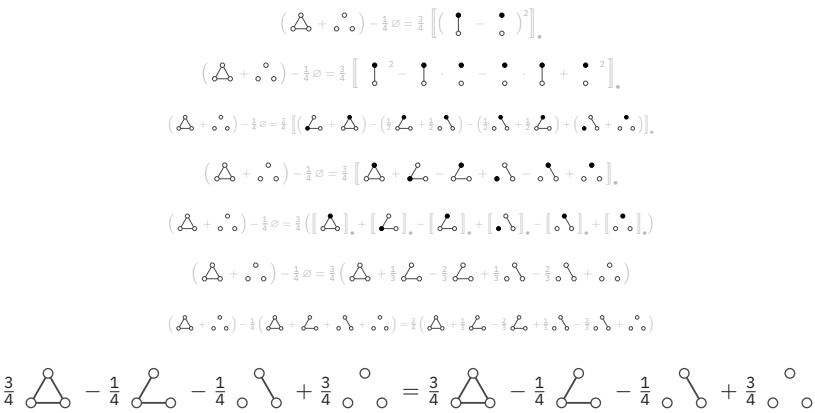


Apply downward operators

A Mechanical Verification of a Flag Algebra Certificate

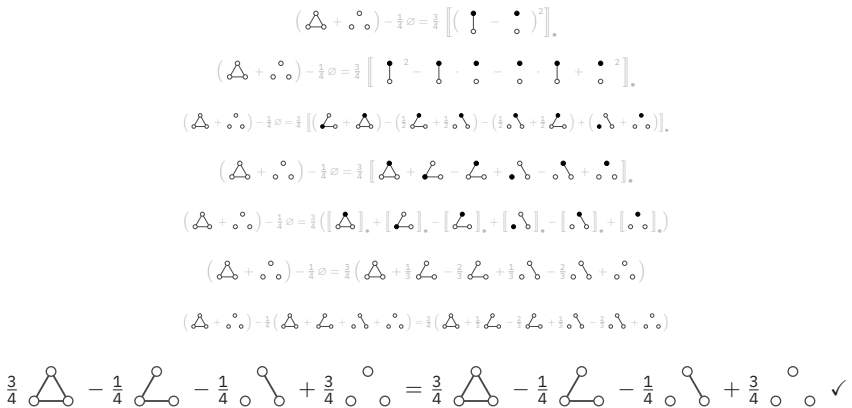
Bring all terms to the smallest common order

A Mechanical Verification of a Flag Algebra Certificate



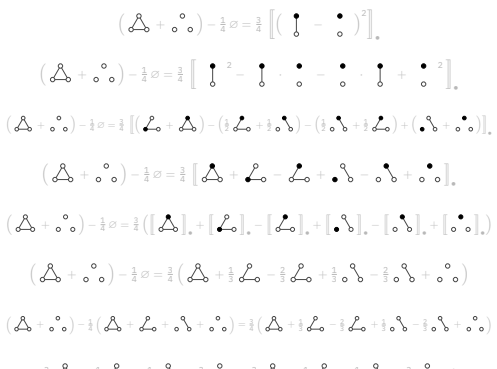
Canonize through commutativity and distributivity

A Mechanical Verification of a Flag Algebra Certificate



Compare coefficients

A Mechanical Verification of a Flag Algebra Certificate



These proofs and their coefficients quickly grow far too large to carry out by hand...

Flag Algebra Certificates in the Wild

… so in practice everyone provides their own code that verifies the certificate for you.

Framework	Active	Structures	Language	Serialization	Certificate
Flagmatic E. Vaughan	2012–14	graphs, 3-graphs, oriented	Python/Cython	JSON	rational PSD
permpack J. Sliačan	2016–18	permutations	SageMath	—	—
Flagmatic (Sage fork) J. Sliačan	2018–22	graphs, 3-graphs, oriented	Python/Cython	JSON	rational PSD
flag-algebra Coregliano, Parente, Sato	2018	tournaments, graphs, digraphs	C++	plain text	PSD / SOS
FlagSOS.jl D. Brosch	2020–25	(hyper)graphs, codes, trees	Julia	JLD2	SOS (rational)
rust-flag-algebra R. de Verclos	2020–25	graphs, digraphs, colored	Rust	SDPA / HTML	PSD (numeric)
FlagAlgebraToolbox L. Bodnár	2025–26	(hyper)graph (coloring)s, digraphs	SageMath	—	rational PSD
C. Spiegel *et al.*	2023–24	graph colorings	Python/Sage	YAML	rational PSD
B. Lidický *et al.*	2014–25	graphs, hypergraphs, …	C++ / Sage	`.dat-s` + `.sage`	per paper
D. Brosch *et al.*	2024–25	rooted trees	Julia	JLD2	SOS (rational)
D. Brosch *et al.*	2025	graph colorings	C++ / Julia	—	SDP + SAT/ILP

What Are We Actually Trusting?

What we put our trust in

- Razborov's Flag Algebra Theory
  product, averaging, Positivstellensatz, ...
- The Bespoke Verification Code
  flag product, downward operator, certificate parsing, ...
- Graph and Combinatorics Libraries
  canonical labeling (`nauty`, Bliss, Traces), isomorph-free generation (`geng`), SageMath, NetworkX, igraph, ...
- Exact Arithmetic
  GMP, FLINT, numpy, LAPACK, ...
- Compiler, Runtime & Toolchain
  GCC, Clang, LLVM, CPython, Julia JIT, ...

What Are We Actually Trusting?

What we put our trust in

- Razborov's Flag Algebra Theory
  product, averaging, Positivstellensatz, ...
- The Bespoke Verification Code
  flag product, downward operator, certificate parsing, ...
- Graph and Combinatorics Libraries
  canonical labeling (`nauty`, Bliss, Traces), isomorph-free generation (geng), SageMath, NetworkX, igraph, ...
- Exact Arithmetic
  GMP, FLINT, numpy, LAPACK, ...
- Compiler, Runtime & Toolchain
  GCC, Clang, LLVM, CPython, Julia JIT, ...

How that trust is usually earned

- The Peer Review Process
  but this rarely covers code

What Are We Actually Trusting?

What we put our trust in

- Razborov's Flag Algebra Theory
  product, averaging, Positivstellensatz, ...
- The Bespoke Verification Code
  flag product, downward operator, certificate parsing, ...
- Graph and Combinatorics Libraries
  canonical labeling (`nauty`, Bliss, Traces), isomorph-free generation (geng), SageMath, NetworkX, igraph, ...
- Exact Arithmetic
  GMP, FLINT, numpy, LAPACK, ...
- Compiler, Runtime & Toolchain
  GCC, Clang, LLVM, CPython, Julia JIT, ...

How that trust is usually earned

- The Peer Review Process
  but this rarely covers code
- Independent Re-implementations
  by other groups, in other languages
- Trusted Dependencies
  old, well maintained, and commonly used
- Integration Test Suites
  against known values, small cases, OEIS, ...
- 'Safe' Languages
  memory and type safety

What Are We Actually Trusting?

What we put our trust in

- Razborov's Flag Algebra Theory
  product, averaging, Positivstellensatz, ...
- The Bespoke Verification Code
  flag product, downward operator, certificate parsing, ...
- Graph and Combinatorics Libraries
  canonical labeling (`nauty`, Bliss, Traces), isomorph-free generation (geng), SageMath, NetworkX, igraph, ...
- Exact Arithmetic
  GMP, FLINT, numpy, LAPACK, ...
- Compiler, Runtime & Toolchain
  GCC, Clang, LLVM, CPython, Julia JIT, ...

How that trust is usually earned

- The Peer Review Process
  but this rarely covers code
- Independent Re-implementations
  by other groups, in other languages
- Trusted Dependencies
  old, well maintained, and commonly used
- Integration Test Suites
  against known values, small cases, OEIS, ...
- 'Safe' Languages
  memory and type safety

The ultimate 'safe' language is one whose type system also checks the mathematics itself.

Formal Proof Verification and Verified Computing in Lean

Two Views of Formal Proof Verification in Lean

1. A Playground for Axiomatic Logic.

You prove and the machine checks down to the axioms. What you end up trusting:

- The playground itself and the usual software / hardware stack.
- The axiomatic system of logic it chooses as its foundation.
- The definitions — your own and those of any library you build on.

Two Views of Formal Proof Verification in Lean

1. A Playground for Axiomatic Logic.

You prove and the machine checks down to the axioms. What you end up trusting:

- The playground itself and the usual software / hardware stack.
- The axiomatic system of logic it chooses as its foundation.
- The definitions — your own and those of any library you build on.

A shared foundational library like `mathlib` buys two things at once: reuse (less work for you) and trust (less chance to screw up your definitions).

Two Views of Formal Proof Verification in Lean

1. A Playground for Axiomatic Logic.

You prove and the machine checks down to the axioms. What you end up trusting:

- The playground itself and the usual software / hardware stack.
- The axiomatic system of logic it chooses as its foundation.
- The definitions — your own and those of any library you build on.

A shared foundational library like `mathlib` buys two things at once: reuse (less work for you) and trust (less chance to screw up your definitions).

▷ *A formally verified graph theoretic statement building on* `mathlib` *notions.*

Two Views of Formal Proof Verification in Lean

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

Assembly
nothing — just bits and opcodes

```
mov  eax, [rbx]  ; int? float? -- CPU does not care
add  eax, 4
```

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

Assembly
nothing — just bits and opcodes

```
mov  eax, [rbx]  ; int? float? -- CPU does not care
add  eax, 4
```

Typed Languages (C, Fortran, ...)
the *kind* of a value

```
int add(int x, int y);
add("hello", 3.14); // error: wrong kinds
```

Two Views of Formal Proof Verification in Lean

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

Assembly
nothing — just bits and opcodes

```
mov  eax, [rbx]  ; int? float? -- CPU does not care
add  eax, 4
```

Typed Languages (C, Fortran, ...)
the *kind* of a value

```
int add(int x, int y);
add("hello", 3.14); // error: wrong kinds
```

Memory-Safe Languages (Rust, Swift, ...)
you cannot touch memory you should not

```
drop(s);
println!("{}", s); // error: moved
```

Two Views of Formal Proof Verification in Lean

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

Assembly
nothing — just bits and opcodes

```
mov  eax, [rbx]  ; int? float? -- CPU does not care
add  eax, 4
```

Typed Languages (C, Fortran, ...)
the *kind* of a value

```
int add(int x, int y);
add("hello", 3.14); // error: wrong kinds
```

Memory-Safe Languages (Rust, Swift, ...)
you cannot touch memory you should not

```
drop(s);
println!("{}", s); // error: moved
```

Dependent Types (Haskell/GADTs, Idris, ...)
a type can carry a property

```
head : Vect (S n) a -> a  -- non-empty by type
head []  -- compile error
```

Two Views of Formal Proof Verification in Lean

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

Assembly
nothing — just bits and opcodes

```
mov  eax, [rbx]  ; int? float? -- CPU does not care
add  eax, 4
```

Typed Languages (C, Fortran, ...)
the *kind* of a value

```
int add(int x, int y);
add("hello", 3.14); // error: wrong kinds
```

Memory-Safe Languages (Rust, Swift, ...)
you cannot touch memory you should not

```
drop(s);
println!("{}", s); // error: moved
```

Dependent Types (Haskell/GADTs, Idris, ...)
a type can carry a property

```
head : Vect (S n) a -> a  -- non-empty by type
head []  -- compile error
```

Curry–Howard (Lean, Coq/Rocq, Agda, ...)
a type is a full mathematical claim

... we have just seen an example of that

Two Views of Formal Proof Verification in Lean

2. A Programming Language With a Souped-up Type System.

How much can the type system promise, at compile time?

Assembly
nothing — just bits and opcodes

```
mov  eax, [rbx]  ; int? float? -- CPU does not care
add  eax, 4
```

Typed Languages (C, Fortran, ...)
the *kind* of a value

```
int add(int x, int y);
add("hello", 3.14); // error: wrong kinds
```

Memory-Safe Languages (Rust, Swift, ...)
you cannot touch memory you should not

```
drop(s);
println!("{}", s); // error: moved
```

Dependent Types (Haskell/GADTs, Idris, ...)
a type can carry a property

```
head : Vect (S n) a -> a  -- non-empty by type
head []  -- compile error
```

Curry–Howard (Lean, Coq/Rocq, Agda, ...)
a type is a full mathematical claim

... we have just seen an example of that

- ▷ *Verifying the handshaking lemma computationally for concrete graphs.*

Curry–Howard Correspondence and What We No Longer Trust

Curry–Howard: propositions *are* types and proofs *are* programs.

The handshaking lemma we *stated* and the one we *ran* are the same object: the very notions `mathlib` uses to *state* mathematics can be used for computations.

Caveat: `Classical` reasoning yields proofs with nothing to *run* — and `mathlib` uses `Classical` and `noncomputable` *a lot*, as we already saw in our trivial example. But it is avoidable!

Curry–Howard Correspondence and What We No Longer Trust

Curry–Howard: propositions *are* types and proofs *are* programs.

The handshaking lemma we *stated* and the one we *ran* are the same object: the very notions `mathlib` uses to *state* mathematics can be used for computations.

Caveat: `Classical` reasoning yields proofs with nothing to *run* — and `mathlib` uses `Classical` and `noncomputable` *a lot*, as we already saw in our trivial example. But it is avoidable!

No longer trusted

- The mathematics
- The bespoke certificate-checking code
- The used dependencies
- The paper-to-code correspondence

Still trusted

- Lean's kernel and toolchain
- The axioms it builds on
- Our and `mathlib`'s definitions

Curry–Howard Correspondence and What We No Longer Trust

Curry–Howard: propositions *are* types and proofs *are* programs.

The handshaking lemma we *stated* and the one we *ran* are the same object: the very notions `mathlib` uses to *state* mathematics can be used for computations.

Caveat: `Classical` reasoning yields proofs with nothing to *run* — and `mathlib` uses `Classical` and `noncomputable` *a lot*, as we already saw in our trivial example. But it is avoidable!

No longer trusted

- The mathematics
- The bespoke certificate-checking code
- The used dependencies
- The paper-to-code correspondence

Still trusted

- Lean's kernel and toolchain
- The axioms it builds on
- Our and `mathlib`'s definitions

▷ *Let's see computational flag algebras in Lean!*

Wrapping Up

Wrapping Up & Outlook

Every component you saw was tied to `mathlib` definitions and statements.
Everything is as rigorously verified as any Lean formalized statement.

Wrapping Up & Outlook

Every component you saw was tied to `mathlib` definitions and statements.
Everything is as rigorously verified as any Lean formalized statement.

Some caveats...

- Far harder than doing the same in a 'normal' language: thousands of lines of code
- Basic computations missing in `mathlib`
- Single fixed-order flag space, not full algebra
- **No bridge to the combinatorial bound yet**
- Algorithms very rudimentary (5 vertices)

Wrapping Up & Outlook

Every component you saw was tied to `mathlib` definitions and statements.
Everything is as rigorously verified as any Lean formalized statement.

Some caveats...

- Far harder than doing the same in a 'normal' language: thousands of lines of code
- Basic computations missing in `mathlib`
- Single fixed-order flag space, not full algebra
- **No bridge to the combinatorial bound yet**
- Algorithms very rudimentary (5 vertices)

Outlook

- Complete algebra to combinatorics bridge
- Optimise computational components
- Push the core computations into `mathlib`
- Abstract the requirements of a flag algebra to easily expand beyond just graphs

Wrapping Up & Outlook

Every component you saw was tied to `mathlib` definitions and statements.
Everything is as rigorously verified as any Lean formalized statement.

Some caveats...

- Far harder than doing the same in a 'normal' language: thousands of lines of code
- Basic computations missing in `mathlib`
- Single fixed-order flag space, not full algebra
- **No bridge to the combinatorial bound yet**
- Algorithms very rudimentary (5 vertices)

Outlook

- Complete algebra to combinatorics bridge
- Optimise computational components
- Push the core computations into `mathlib`
- Abstract the requirements of a flag algebra to easily expand beyond just graphs

Code not yet public, hopefully will have a cleaned up version ready by end of year...

Thank you!