Trace Elimination as Closed-Form Discovery: A Duality Between Dynamical Systems and Programs

Erik Arne Mathiesen-Dreyfus

Gimle Labs, www.gimlelabs.com

Abstract

We identify a structural correspondence between two traced monoidal categories: the category of signal-flow circuits used to represent dynamical systems (where trace implements feedback and fixed-point iteration), and the category of sequential programs (where trace implements loops and recursion). In both categories, a *trace-free* morphism corresponds to a direct, non-iterative computation—a closed-form solution on the continuous side, a straight-line program on the discrete side. We show that the general Hoare-logic framework of Arthan, Martin, Mathiesen, and Oliva [1] already provides the formal bridge: both categories admit verification functors to a common Hoare category that preserve the traced monoidal structure, so that trace elimination in one corresponds to trace elimination in the other. With this bridge established, we develop the consequences. We characterise "closed-form solvability" as a definability property, the existence of an explicit witness in a designated elementary subalgebra, and show this characterisation is uniform across both categories. We then argue that trace elimination is undecidable in general by reduction from the halting problem, formalising the classical intuition that "most ODEs don't have closed-form solutions" and connecting it to the undecidability of loop elimination in programs.

1 Introduction

Two fields reason about systems with feedback using strikingly similar mathematical structures, apparently without recognising the connection.

In the study of dynamical systems, signal-flow circuits provide a compositional representation: atomic operations (integration, differentiation, addition, multiplication) are wired together using sequential composition, parallel composition, and a feedback operator called *trace*. These circuits form a traced monoidal category [4]. The trace captures the recursive structure of differential equations: $f = f_0 + \int f\,dx$ is a fixed-point equation, and the corresponding circuit uses trace to feed the output back through the integrator.

In the study of programs, Hoare logic and its generalisations reason about sequential composition, parallel composition, and loops. Martin, Mathiesen, and Oliva [5] showed that this reasoning can be lifted to an abstract traced symmetric monoidal category, where trace corresponds to loop iteration and the categorical axioms yield sound and complete proof rules. Arthan, Martin, Mathiesen, and Oliva [1] then demonstrated that this framework applies uniformly to both program verification *and* stream circuit verification, constructing explicit verification functors for each.

Both settings have a notion of "solving" or "simplifying" a traced morphism: on the continuous side, finding a closed-form solution means eliminating the feedback loop; on the discrete side, loop elimination or unrolling means replacing iterative computation with direct computation. In both cases, the result is a *trace-free* morphism.

This note makes three contributions. First, we observe that the framework of [1] already establishes the formal bridge between these two categories, turning what might appear to be a conjecture into a consequence of existing results (Section 4). Second, we develop a precise characterisation of when trace elimination is possible, connecting it to the existence of closed-form expressions in designated subalgebras (Section 6). Third, we show that trace elimination is undecidable in general by reduction from the halting problem, and explore what this means for the classical question of closed-form solvability of ODEs (Section 7).

2 Two Traced Monoidal Categories

2.1 The continuous side: signal-flow circuits

Let **Circ** denote the traced monoidal category of signal-flow circuits, as realised in Asgard [3]. Objects are natural numbers (wire counts / degrees). Morphisms $f : m \to n$ are circuits with m input wires and n output wires, built from:

- **Atomic operations**: generators ($0 \to 1$: constants, parameters, and fundamental/transcendental functions such as $\exp$, $\sin$, $\cos$, $\log$), transformers ($1 \to 1$: `register`, `deregister`, `scalar`), binary operations ($2 \to 1$: addition, Cauchy product), and routing (`split`, `swap`, `terminal`).
- **Composition**: $f \circ g$ (sequential), $f \otimes g$ (parallel/monoidal).
- **Trace**: $\mathrm{Tr}(f) : m \to n$ for $f : m + 1 \to n + 1$, feeding the last output back to the first input.

Under the real calculus, streams are Taylor coefficient sequences and `register` is formal integration. Trace implements Picard iteration: the fixed-point equation $Y = F(X, Y)$ is solved coefficient-by-coefficient.

A *closed-form solution* is a circuit that computes the same function as a traced circuit but uses no trace. It may use special functions (`exp`, `sin`, etc.) as atomics, but it contains no feedback.

2.2 The discrete side: sequential programs

Let **Prog** denote the traced monoidal category of sequential programs, following the construction in [5, 1]. Objects are state spaces (or types). Morphisms are state transformers. The categorical structure maps to familiar programming constructs:

- **Atomic operations**: standard programming primitives—arithmetic ($+$, $\times$, etc.), conditionals (`if`/branching), joins (`merge`), assignments, and calls to library functions.
- **Composition**: sequential execution ($f; g$) and disjoint union ($f + g$), representing branching over sum types.
- **Trace**: loop iteration — $\mathrm{Tr}(f)$ is the program that repeatedly applies f, feeding part of the output back as input, until a termination condition is met.

Hoare triples $\{P\}\, f\, \{Q\}$ express partial correctness: if precondition P holds before executing f, then postcondition Q holds after. The categorical axioms (naturality, vanishing, superposing for trace) correspond to proof rules for while-loops.

A *straight-line program* (or loop-free program) is a morphism that uses composition and monoidal product but no trace. It computes a direct mapping from input to output without iteration.

3 The Structural Parallel

The structural correspondence is:

	Circ (continuous)	**Prog** (discrete)
Objects	Wire counts (degrees)	State spaces (types)
Morphisms	Signal-flow circuits	State transformers
Composition	Sequential wiring	Sequential execution
Monoidal	Parallel wiring	Disjoint union (branching)
Trace	Feedback / fixed-point	Loop / iteration
Trace-free morphism	Closed-form solution	Straight-line program
Trace elimination	Solving the ODE	Loop elimination
Verification	Stream circuit Hoare logic	Program Hoare logic

The key observation: in both categories, the "hard" morphisms are those with trace (feedback/loops), and the "solved" morphisms are trace-free (closed-form/straight-line). The process of going from one to the other—if it exists—is a proof in the respective rewrite system.

4 The Bridge: Verification Functors and the Hoare Category

What might initially seem like a loose analogy turns out to be a formal correspondence, established by the framework of [1].

4.1 The Hoare category

Let $\mathbf{H}$ denote the *Hoare category*: objects are pre-ordered sets, and morphisms are monotone relations. The monoidal product is the cartesian product, and the trace is defined by existential quantification over the feedback variable:

$$x \ \mathrm{Tr}(r) \ y \quad \Longleftrightarrow \quad \exists z.\ (x, z)\ r\ (y, z).$$

This makes $\mathbf{H}$ a traced symmetric monoidal category.

4.2 Verification functors

A *verification functor* [1] is a strict traced monoidal functor $H : \mathbf{S} \to \mathbf{H}$ from a semantic category $\mathbf{S}$ to the Hoare category. The key properties are:

1. H maps sequential composition to relational composition: $H(f \circ g) = H(g) \circ H(f)$.
2. H maps monoidal product to cartesian product: $H(f \otimes g) = H(f) \times H(g)$.
3. H maps trace to trace: $H(\mathrm{Tr}(f)) = \mathrm{Tr}(H(f))$.

The main theorem of [1] is that any verification functor automatically yields a sound and complete Hoare logic. The proof rules for sequential composition, parallel composition, and loops follow directly from the functorial properties.

4.3 The bridge

The critical observation is that [1] constructs verification functors for *both* of our categories:

- $H_{\mathrm{Prog}} : \mathbf{Prog} \to \mathbf{H}$, using strongest postconditions (or weakest liberal preconditions). This yields the classical Hoare logic for while-programs, and via a different choice of concrete category, separation logic for pointer programs. A third instantiation yields a novel Hoare logic for running-time analysis.
- $H_{\mathrm{Circ}} : \mathbf{Circ} \to \mathbf{H}$, using the algebraic properties of formal power series. This yields a Hoare logic for stream circuits, where the trace axiom corresponds to solving the fixed-point equation coefficient-by-coefficient.

  Both functors land in the *same* target category $\mathbf{H}$. This gives us:

Theorem 1 (Trace-elimination correspondence)**.** *The verification functors H_{Prog} and H_{Circ} provide a span*

$$\mathbf{Prog} \xrightarrow{H_{\mathrm{Prog}}} \mathbf{H} \xleftarrow{H_{\mathrm{Circ}}} \mathbf{Circ}$$

in which:

1. *Both functors preserve the traced monoidal structure.*
2. *Trace-free morphisms map to relations that do not use existential quantification over feedback variables.*
3. *A trace-elimination rewrite on either side (a derivation showing that a traced morphism is semantically equivalent to a trace-free one) maps, via the respective verification functor, to a trace elimination in $\mathbf{H}$: the existential quantifier in $\mathrm{Tr}_{\mathbf{H}}$ is resolved by a direct witness. In this sense, both sides project to the same problem in $\mathbf{H}$, though the span does not provide a direct lift from one source category to the other.*

Proof sketch. Properties (1) and (2) follow directly from the definition of verification functor and the construction in [1]. For (3), observe that in $\mathbf{H}$, the trace $\mathrm{Tr}(r)$ is defined by $\exists z.\ (x, z)\ r\ (y, z)$. A trace-free morphism in $\mathbf{H}$ is a relation s such that $x\ s\ y$ holds without quantifying over any auxiliary variable z. Trace elimination in $\mathbf{H}$ thus amounts to showing that the existential can be resolved—that a definite witness $z = \varphi(x)$ (or $z = \varphi(y)$) exists and can be substituted. Since both verification functors preserve trace, a trace-elimination rewrite $\mathrm{Tr}(f) = g$ (with g trace-free) in either source category maps to $\mathrm{Tr}(H(f)) = H(g)$ in $\mathbf{H}$, which is the same existential-elimination problem regardless of which source category it came from. □

Remark 1. *The span through $\mathbf{H}$ is not a direct functor $\mathbf{Circ} \to \mathbf{Prog}$, but it is sufficient for our purposes: it provides a common semantic domain in which trace elimination has a uniform characterisation. Two morphisms—one a circuit, one a program—are "the same problem" if their images in $\mathbf{H}$ coincide.*

5 Bainbridge Duality and Coalgebra

The bridge through $\mathbf{H}$ can be understood in terms of the coalgebra/algebra duality identified by Bainbridge [2] and refined by Rutten [6, 7] and Turi–Plotkin [8]. Traced morphisms are *coalgebraic*: they describe systems dynamically, via feedback and state transitions. Trace-free morphisms are *algebraic*: they describe the same systems statically, via direct expressions. Trace elimination is the passage from coalgebra to algebra, and this passage has the same structure on both sides of our correspondence.

In $\mathbf{Circ}$, a traced circuit [3] defines a dynamical system by feedback through an integrator; its trace-free equivalent (if one exists) is a closed-form expression using elementary functions. In $\mathbf{Prog}$, a loop defines a computation by iterated state update; its trace-free equivalent is a straight-line program that computes the result directly. The compiler optimisations of "loop strength reduction" and "closed-form loop elimination" are informal versions of this same coalgebra-to-algebra passage.

6 Closed-Form Solvability as a Definability Property

With the bridge established, we can ask: *when* does trace elimination succeed? This question has a uniform answer across both categories.

Definition 1 (Elementary subalgebra)**.** *Let $\mathbf{C}$ be a traced monoidal category with a designated set $\mathcal{E}$ of elementary morphisms—the trace-free atomics from which "closed-form" expressions are built. The elementary subalgebra $\mathbf{C}_{\mathcal{E}}$ is the wide sub-PRO (same objects, restricted morphisms) consisting of all morphisms that can be expressed as finite compositions and monoidal products of elements of $\mathcal{E}$, without using trace.*

In $\mathbf{Circ}$, the elementary morphisms are the standard special functions: $\exp$, $\sin$, $\cos$, polynomials, and their compositions under the ring operations. In $\mathbf{Prog}$, they are the primitive operations of the programming language: arithmetic, conditionals (as multiplexers), and function calls to a standard library.

Definition 2 (Trace eliminability)**.** *A traced morphism $\mathrm{Tr}(f)$ in $\mathbf{C}$ is trace-eliminable (relative to $\mathcal{E}$) if there exists a morphism $g \in \mathbf{C}_{\mathcal{E}}$ such that $\mathrm{Tr}(f)$ and g are semantically equivalent, i.e., they denote the same morphism under the interpretation: $[\![\mathrm{Tr}(f)]\!] = [\![g]\!]$.*

This definition makes precise what "closed-form solution" means: it is not an intrinsic property of the dynamical system, but a property *relative to a choice of elementary functions*. The ODE $f' = f$ has a closed-form solution relative to $\{\exp\}$ (namely $f = ce^x$) but not relative to $\{+, \times, \text{polynomials}\}$ alone.

Proposition 1 (Uniform characterisation via $\mathbf{H}$)**.** *Let $\mathrm{Tr}(f)$ be a traced morphism in $\mathbf{Circ}$ or $\mathbf{Prog}$, and let $r = H(\mathrm{Tr}(f)) = \mathrm{Tr}(H(f))$ be its image in $\mathbf{H}$. Then $\mathrm{Tr}(f)$ is trace-eliminable if and only if the existential quantifier in r,*

$$x \; r \; y \iff \exists z.\ (x, z)\ H(f)\ (y, z),$$

can be resolved by an explicit witness $z = \varphi(x)$ where φ is expressible in the image of the elementary subalgebra under H.

Proof sketch. For the forward direction, suppose $\mathrm{Tr}(f)$ is trace-eliminable, i.e., there exists a trace-free $g \in \mathbf{C}_{\mathcal{E}}$ with $[\![\mathrm{Tr}(f)]\!] = [\![g]\!]$. Since H preserves trace, $H(\mathrm{Tr}(f)) = \mathrm{Tr}(H(f))$. Since H preserves composition and monoidal product, $H(g)$ lies in the image of $\mathbf{C}_{\mathcal{E}}$ and is trace-free in $\mathbf{H}$. The equality $\mathrm{Tr}(H(f)) = H(g)$ means the existential $\exists z.\ (x, z)\ H(f)\ (y, z)$ is equivalent to the direct relation $x\ H(g)\ y$. The witness $z = \varphi(x)$ is extracted from the structure of g: since g computes a definite output without feedback, the value of the internal wire that g "eliminates" is determined as a function of the input.

For the converse, the existence of an elementary witness φ in $\mathbf{H}$ does not automatically lift to a trace-free morphism in the source category. The converse therefore requires an additional assumption: that the verification functor is sufficiently faithful, i.e., that the elementary subalgebra of $\mathbf{C}$ maps injectively into $\mathbf{H}$. Under this assumption, the witness φ in $\mathbf{H}$ corresponds to a unique trace-free morphism in $\mathbf{C}_{\mathcal{E}}$. □

This reframes closed-form solvability as a question about *definability*: can the implicit function defined by the fixed-point equation be made explicit using a given vocabulary? On the circuit side, this is classical: the ODE $f' = f$ defines $\exp$ implicitly, and "solving" it means recognising that $\exp$ is in our vocabulary. On the program side, this is the question of whether a loop's accumulator has an explicit formula as a function of the iteration count.

6.1 The role of the elementary subalgebra

The choice of $\mathcal{E}$ is crucial and often implicit in classical discussions of solvability:

- **Liouville's theorem** (1835): the indefinite integral $\int e^{-x^2} dx$ is not expressible in terms of elementary functions (polynomials, exponentials, logarithms, and their compositions). Here $\mathcal{E}$ is the classical elementary functions.
- **Differential Galois theory**: characterises when a linear ODE has solutions in a given differential field extension. The "closed-form" question is always relative to the base field.
- **Loop optimisation**: a compiler can eliminate the loop `for i in 1..n: s += i` if it knows the formula $s = n(n+1)/2$, which requires quadratic polynomials in $\mathcal{E}$. But `for i in 1..n: s += f(i)` for arbitrary f cannot in general be eliminated.

In each case, the question is the same: does the implicit function defined by the trace (feedback/loop) lie in the elementary subalgebra? The duality through $\mathbf{H}$ ensures this question has the same structure on both sides.

7 Undecidability of Trace Elimination

We now turn to the negative side: when trace elimination is *impossible*, and what this impossibility means.

7.1 From the halting problem to trace elimination

Theorem 2 (Undecidability of trace elimination)**.** *The problem "given a traced morphism $\mathrm{Tr}(f)$ in $\mathbf{Prog}$, does there exist a trace-free morphism g such that $[\![\mathrm{Tr}(f)]\!] = [\![g]\!]$?" is undecidable.*

Proof sketch. We reduce from the halting problem. Let M be a Turing machine and w an input. For each pair (M, w), construct a morphism $f_{M,w} : 1 + S \to 1 + S$ in $\mathbf{Prog}$ (where S is the state space of M) that performs one step of M's computation on w: if M has not yet halted, it outputs the updated state into the right summand S (feeding back via trace); if M has halted, it outputs a fixed value v into the left summand 1 (exiting the loop). The traced morphism $\mathrm{Tr}(f_{M,w}) : 1 \to 1$ simulates the entire computation.

If M halts on w, then $[\![\mathrm{Tr}(f_{M,w})]\!]$ is the constant function returning v, which is the denotation of a trace-free morphism (a straight-line program that simply returns v).

If M does not halt on w, then $[\![\mathrm{Tr}(f_{M,w})]\!]$ is the undefined partial function $\bot : 1 \to 1$ (the loop never exits). Since trace-free morphisms in $\mathbf{Prog}$ are built from finite compositions of total atomics, they are necessarily total. No total morphism can be semantically equivalent to $\bot$.

Thus $\mathrm{Tr}(f_{M,w})$ is trace-eliminable if and only if M halts on w. □

Remark 2. *The argument requires that $\mathbf{Prog}$ contains partial morphisms (which it does, since trace with disjoint-union monoidal product naturally models partial functions via non-termination). Whether undecidability persists in a purely total setting requires a separate argument and is left as an open question.*

7.2 Transfer to the continuous side

The bridge through $\mathbf{H}$ now yields:

Corollary 1 (Undecidability of closed-form solvability)**.** *If $\mathbf{Circ}$ is sufficiently expressive to encode the computations of $\mathbf{Prog}$ (i.e., if the verification functor H_{Circ} has an image that contains the image of H_{Prog} on the relevant morphisms), then the problem "given a traced circuit $\mathrm{Tr}(f)$ in $\mathbf{Circ}$, does there exist a trace-free circuit g such that $[\![\mathrm{Tr}(f)]\!] = [\![g]\!]$?" is also undecidable.*

The expressiveness condition is plausible: the Asgard circuit language [3], interpreted under the discrete calculus, can simulate arbitrary recurrences, and hence arbitrary computations. The real calculus side is more subtle—it corresponds to analytic functions, and the question becomes whether the class of analytic ODEs is rich enough to encode arbitrary computation.

7.3 What undecidability means for ODEs

The classical theory already contains strong negative results:

- **Richardson's theorem** (1968): it is undecidable whether a given expression involving $+$, $\times$, $\sin$, $\exp$, $|\cdot|$, π, $\ln 2$, and composition is identically zero. This means that even *verifying* a proposed closed-form solution is undecidable in general.
- **Differential Galois theory**: provides decidable criteria for *second-order linear* ODEs (the Kovacic algorithm, 1986), with extensions to higher-order linear equations via related methods. The general nonlinear case remains open.
- **Pour-El and Richards** (1981): showed that the three-dimensional wave equation can have computable initial data but non-computable solutions at a specific time point, demonstrating that continuous dynamics can amplify computational complexity.

Our result adds a categorical perspective: the undecidability of trace elimination in $\mathbf{Prog}$ (a well-understood consequence of the halting problem) *transfers* to $\mathbf{Circ}$ via the common Hoare category. This is not merely an analogy—it is a formal transfer along a structure-preserving span of functors.

By analogy (not yet formalised), the transfer suggests a hierarchy of solvability reminiscent of the arithmetic hierarchy:

- **Trace-free**: the morphism is already in the elementary subalgebra. No fixed-point iteration is needed. (Closed-form solutions, straight-line programs.)
- **Trace-eliminable**: the morphism uses trace, but can be rewritten to a trace-free equivalent. The fixed point exists and is expressible. (Solvable ODEs, eliminable loops.)
- **Provably non-eliminable**: it can be proved that no trace-free equivalent exists. (ODEs with no elementary closed form, provably infinite loops.)

Theorem 2 shows that the general problem of determining which level a morphism belongs to is undecidable in $\mathbf{Prog}$. Making this hierarchy precise, in particular, whether it admits a grading by quantifier complexity analogous to the arithmetic hierarchy, is an open problem.

8 Cut-Elimination as Trace-Elimination

A third instance of the same structural pattern arises in proof theory, and it sharpens the correspondence considerably.

8.1 The cut rule is a trace

In Gentzen's sequent calculus, the *cut rule* combines two proofs, one establishing $\Gamma \vdash \Delta, A$ and the other $A, \Gamma' \vdash \Delta'$, into a single proof of $\Gamma, \Gamma' \vdash \Delta, \Delta'$, with the formula A no longer appearing. Read categorically, this is exactly a trace: the two proofs are placed side by side using the monoidal product (juxtaposing their sequents), and the trace operator then identifies the occurrence of A on the right of the first proof with the occurrence of A on the left of the second, "cutting away" the shared formula by feeding the conclusion of one into the hypothesis of the other.

Concretely, if proofs are morphisms in a traced monoidal category of sequents, then

$$\mathrm{cut}_A(\pi_1, \pi_2) \;=\; \mathrm{Tr}^A\big(\pi_1 \otimes \pi_2\big),$$

where the trace is taken over the wire labelled A. The cut formula is the feedback variable; the cut rule is its elimination as a free variable from the joint proof.

8.2 Cut-elimination as analytical proofs

Gentzen's *Hauptsatz* (cut-elimination theorem) shows that any proof using cut can be transformed into a cut-free proof of the same sequent. Cut-free proofs are called *analytical*: they exhibit a direct, witness-by-witness derivation in which every formula appearing in the proof is a subformula of the endsequent (the subformula property). They are, in our terminology, the trace-free morphisms of the proof category.

The parallel to the previous sections is now exact:

$$\text{cut-elimination} \;:\; \text{cut} \longmapsto \text{cut-free proof} \quad\equiv\quad \text{trace-elimination} \;:\; \mathrm{Tr}(f) \longmapsto g \in \mathbf{C}_{\mathcal{E}}.$$

An analytical proof is to a logical theorem what a closed-form solution is to a differential equation: a direct, witness-providing derivation that uses no auxiliary lemma to be "cut away."

8.3 Curry–Howard: strong normalisation as termination

Under the Curry–Howard correspondence, sequent calculus proofs correspond to typed lambda terms, and cut-elimination corresponds to β-reduction. A proof is cut-free precisely when the corresponding term is in normal form. Gentzen's Hauptsatz becomes the statement that every well-typed term reduces to a normal form, i.e., *strong normalisation*.

Strong normalisation in turn is exactly the guarantee of *termination*: a programming language whose type system enforces strong normalisation (e.g., the simply typed lambda calculus, System F, or a total dependent type theory) admits only programs that terminate, and therefore admits, in principle, a closed-form unfolding of every expression. The trace-free image is computable and finite.

We thus obtain a four-way correspondence:

$\mathbf{Circ}$	$\mathbf{Prog}$	Sequent calculus	Lambda calculus
trace (feedback)	loop / iteration	cut rule	β-redex
trace-free morphism	straight-line program	cut-free (analytical) proof	normal form
trace elimination	loop elimination	cut-elimination	normalisation
closed-form solution	terminating computation	analytic proof	strong normalisation

In each column, the same structural move, removing the trace, corresponds to making explicit what was previously implicit: an answer instead of an equation, a value instead of a loop, a derivation instead of an appeal to a lemma, a result instead of a redex.

8.4 The halting problem breaks the symmetry

The correspondence is clean precisely in the strongly-normalising fragment, where every term has a normal form and every traced morphism is trace-eliminable. Outside that fragment, the symmetry breaks: in a Turing-complete programming language, not every program terminates, not every loop can be eliminated, and, by the halting problem, one cannot decide which can. This is the source of the undecidability we exploited in Theorem 2.

The picture this leaves us with is sharp. *Within* a strongly-normalising calculus, trace-elimination is total: every fixed point is solvable, every loop unrolls, every proof reduces to an analytical one, every dynamical system has a closed form. *Outside* it, in the realm of general computation, of nonlinear ODEs, of arbitrary recursion, trace-elimination is at best a partial operation, and deciding when it succeeds is exactly as hard as deciding halting. The classical unsolvability of most differential equations and the undecidability of the halting problem are, in this view, the same phenomenon viewed through different verification functors into $\mathbf{H}$.

This trade-off is fundamental and runs in both directions. By the halting problem, a calculus that guarantees strong normalisation *cannot* be Turing-complete: there exist computable functions that no strongly-normalising language can express. The simply typed lambda calculus, System F, Gödel's System T, and total dependent type theories all sit on the strong-normalisation side of this divide and pay for their guaranteed termination by surrendering universal expressiveness. Closed-form solvability is bought at the price of computational strength.

8.5 Which dynamical systems are strongly normalisable?

This trade-off raises a natural question on the continuous side. *Which subcategories of* $\mathbf{Circ}$ *are "strongly normalisable"* — in the sense that every traced morphism in the subcategory admits a trace-free equivalent in a designated elementary subalgebra? Such subcategories are the dynamical analogues of total programming languages: expressively limited, but with guaranteed closed-form solutions.

Several classical answers populate this hierarchy:

- **Linear ODEs with constant coefficients**: solvable in closed form via matrix exponentials. The corresponding sub-PRO of $\mathbf{Circ}$ — linear circuits over a finite atomic basis closed under $\exp$ — is strongly normalisable.
- **Linear ODEs with rational coefficients**: the Kovacic algorithm (1986) decides Liouvillian solvability for second-order equations; higher-order extensions exist. Strong normalisation here is decidable but not automatic.
- **Polynomial vector fields with nilpotent linearisation**: integrable in finitely many steps via formal power series truncation.
- **Hamiltonian systems with sufficiently many independent integrals of motion** (Liouville–Arnold integrable systems): admit closed-form quadratures.

In each case, restricting the atomic basis or the allowed combinator structure of $\mathbf{Circ}$ yields a sub-PRO on which trace-elimination is total. This mirrors, on the continuous side, the way restricting lambda calculus to its simply typed fragment yields strong normalisation.

The general programme is to characterise the maximal sub-PROs of $\mathbf{Circ}$ on which trace-elimination is decidable (analogue of total but with decidable type-checking), or total (analogue of strongly normalising), or merely semi-decidable (analogue of Turing-complete with undecidable halting). Such a characterisation would give a structural account of *which* dynamical systems we can hope to solve, and would frame solver design, whether classical (Kovacic, differential Galois) or learned (the Mimir foundation model for circuit synthesis), as the search for the largest tractable sub-PRO appropriate to a given application.

9 Conclusion

We have shown that the structural parallel between two traced monoidal categories—one for dynamical systems, one for programs—is not merely an analogy but a formal correspondence mediated by the Hoare category of [1]. Both categories admit verification functors that preserve the traced monoidal structure, and trace elimination in one corresponds to trace elimination in the other.

With this bridge in place, we developed two consequences. First, we characterised closed-form solvability as a *definability* property: a traced morphism is solvable if and only if the implicit function defined by its fixed point lies in a designated elementary subalgebra. This characterisation is uniform across both categories and connects to classical results in differential Galois theory and compiler optimisation.

Second, we showed that trace elimination is undecidable in $\mathbf{Prog}$ by reduction from the halting problem. The transfer to $\mathbf{Circ}$ via the Hoare category is conditional on an expressiveness assumption (Corollary 1), but if it holds, it formalises the intuition that "most ODEs don't have closed-form solutions" as a consequence of the undecidability of the halting problem.

Several questions remain open. First, whether undecidability persists in a purely total setting, where the halting-based argument does not apply directly. Second, whether the expressiveness condition for the transfer to $\mathbf{Circ}$ can be established—this amounts to showing that the circuit language under the discrete calculus can encode arbitrary computation, which is plausible but unproved. Third, whether the span through $\mathbf{H}$ can be tightened to a direct functor $\mathbf{Circ} \to \mathbf{Prog}$ or vice versa, which would give a stronger correspondence than the current indirect bridge.

The deeper message is that trace—whether as feedback, recursion, or iteration—is the universal mechanism for systems that refer to themselves. Eliminating it is the universal notion of "solving." The impossibility of eliminating it in general connects the classical unsolvability of most differential equations to the undecidability of the halting problem, not by analogy but by categorical structure.

References

[1] Rob Arthan, Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. A general framework for sound and complete floyd–hoare logics. *ACM Transactions on Computational Logic*, 11(1):7:1–7:31, 2009.

[2] Edwin S. Bainbridge. *Feedback and Generalized Logic*. PhD thesis, University of Michigan, 1976.

[3] Gimle Labs. Asgard: A typed compositional framework for dynamical systems. `https://asgard.gimlelabs.com`, 2025.

[4] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[5] Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. Hoare logic in the abstract. In Zoltán Ésik, editor, *Computer Science Logic (CSL 2006)*, volume 4207 of *Lecture Notes in Computer Science*. Springer, 2006.

[6] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[7] Jan J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[8] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 280–291. IEEE, 1997.