The Structure–Logic–Computation (SLC) Framework: Deriving Logic and Computation from Algebraic Structure

Martila Research
research@martila.io

March 10, 2026

Abstract

Given an equational presentation (Σ, E)—a signature with equations—the well-known equivalences between equational derivability, rewrite convertibility, and model validity (Birkhoff completeness) provide a three-way bridge linking algebraic structure, logic, and computation. We present the *Structure–Logic–Computation (SLC) framework*, a machine-executable pipeline that makes these classical equivalences auditable and constructive. From a presentation (Σ, E) we construct (i) an equational-logic category L_T where provability is morphism equality, (ii) a term category C_T whose morphisms are raw Σ-term tuples, together with rewrite graphs on its hom-sets, and (iii) bidirectional translations between equational derivations and traced rewrite certificates, with functorial transport along theory morphisms. A Racket prototype realizes the pipeline with multiple search strategies, preliminary Knuth–Bendix support, and certificates structured to support independent checking. We discuss implications for domain-specific language design and verification.

1 Introduction

The Curry–Howard correspondence reveals [35] a deep isomorphism between proofs and programs, propositions and types. Lambek [19] pushed this further: logical derivations and typed terms appear as morphisms in a cartesian closed category, yielding the Curry–Howard–Lambek (CHL) triad. However, both logic and computation rest on a more basic layer: the *algebraic structure* of the domain.

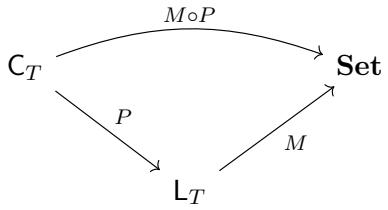

Figure 1: The free term category C_T (raw Σ-terms, no equations) projects onto the quotient L_T (terms modulo provable equality) via P. Any model $M : \mathsf{L}_T \to \mathbf{Set}$ lifts to $\mathsf{C}_T \to \mathbf{Set}$ by composition $M \circ P$. The rewrite relation $\leftrightarrow^*$ is external structure on the hom-sets of C_T, not categorical morphisms; P identifies exactly those morphisms related by $\leftrightarrow^*$.

We systematize these relations as the *SLC framework*: given an *equational presentation* (Σ, E)—a signature of operations with equational axioms—we algorithmically realize the derivation of both

equational proof theory and rewrite-based computation, with bidirectional certificate translation. The presentation induces a Lawvere theory (the quotient category), but the *raw term structure* and *rewrite dynamics* depend on the chosen presentation, not just the abstract theory. The move echoes how CHL has been lifted to concurrency via propositions-as-session-types [7, 34].

Aspect	**CHL (classic)**	**SLC (this work)**
Starting point	Fixed calculus (λ-calc, etc.)	Presentation (Σ, E) of a Lawvere theory
Logic extracted	Typing / proof system	L_T (equational)
Computation extracted	Operational semantics	C_T with rewrite graph (conversion)
Core correspondence	Types $\leftrightarrow$ propositions $\leftrightarrow$ categorical structure	Provability $\leftrightarrow$ convertibility $\leftrightarrow$ model validity

Table 1: CHL versus SLC at a glance.

Why an algebra-first companion is useful. The CHL correspondence presupposes a *fixed* calculus (typically the simply-typed λ-calculus) and thus hard-wires proof theory to one particular notion of program. Modern programming languages and security frameworks continually introduce new primitives—effects, capabilities, resource types—which shift both proof theory and operational semantics.

The SLC framework shifts the starting point to algebraic presentations: given *only* an equational presentation of the domain, it *derives* both the logic (equational provability) and the machine of executions (rewrite paths). This flexibility is particularly useful when the calculus itself is under construction, such as in domain-specific languages or policy-driven runtimes.

Precise relationship to CHL. CHL identifies three things: (i) propositions/types, (ii) proofs/terms, (iii) cartesian-closed categories. SLC identifies a different triple: (i) equational provability in L_T, (ii) rewrite convertibility in the rewrite graph over C_T, (iii) semantic validity in all models. The two correspondences are *complementary*, not competing: CHL applies to *higher-order* calculi with function types ($A \to B$) and inhabitation questions ("is type A inhabited?"); SLC applies to *first-order* algebraic theories with equality questions ("does $s = t$ hold?"). One can view SLC as the "algebraic fragment" that sits beneath CHL: when a simply-typed λ-calculus is equipped with equational axioms (e.g., $\beta\eta$-laws), SLC governs the equational reasoning while CHL governs the typing and proof-term structure. There is no canonical equivalence identifying Lawvere theories with cartesian closed categories; the relation between SLC and CHL is one of structural analogy and possible completion constructions, not direct identification.

Contributions.

- A precise statement of the SLC framework and its place in the Curry–Howard–Lambek lineage, synthesizing classical results (Birkhoff completeness, term rewriting) into a unified framework.
- A constructive proof (Section 4) that each equational presentation (Σ, E) yields an equational-logic category L_T and a term category C_T with induced rewrite dynamics, where convertibility is sound and complete for provable equality.
- A monoid running example grounding the construction.

- Discussion of consequences for domain-specific language design, formal verification, and security, and a roadmap for future work.
- A prototype implementation in Racket [8] that, given an equational presentation (Σ, E), generates an interactive equality checker for L_T and a symbolic rewrite simulator for the rewrite graph over C_T, with bidirectional translation between equational derivations and traced certificates.

Scope of novelty. The mathematical equivalences (Birkhoff completeness, conversion = provability) are classical. Our contribution is *packaging them as a constructive, auditable pipeline*: given any equational presentation (Σ, E), we algorithmically derive an equational prover and a trace-generating rewrite simulator with (i) bidirectional certificate translation, (ii) functorial transport along theory morphisms, and (iii) certificates structured to support independent checking. We treat the underlying mathematics as established and focus on auditability and algorithmic realization.

2 Background

This section recalls the minimal categorical logic needed for the SLC framework. We follow the terminology of Mac Lane [22] (categories), Lawvere [20] (algebraic theories) and Baader–Nipkow [3] (term rewriting). Readers who prefer a one-page intuition before the formalism may glance at Fig. 2. Enriched and indexed variants of Lawvere theories extend the framework to side–effects, names, and local state [25, 10, 31].

2.1 Lawvere theories and presentations

A *Lawvere theory* [20] is a small category L with finite products together with a strict finite-product-preserving identity-on-objects functor $I : \mathbb{F}^{\mathrm{op}} \to L$, where $\mathbb{F}$ is a skeleton of finite sets. Equivalently, one may work skeletally and regard the objects as natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$, with n interpreted as the n-fold product of a distinguished object. For the syntactic Lawvere theory presented by (Σ, E), a morphism $n \to m$ can be represented by an m-tuple $(t_1, \dots, t_m)$ of Σ-terms in n variables, modulo provable equality. A *model* of L in a category $\mathcal{C}$ with finite products is a finite-product-preserving functor $L \to \mathcal{C}$ [16].

Presentations. A Lawvere theory is typically *presented* by a signature Σ (operation symbols with arities) and a set E of equations between Σ-terms [16]. Different presentations can yield the same theory (up to equivalence), but the *term category* C_T and the *rewrite dynamics* depend on the chosen presentation. In our implementation, the user provides a presentation (Σ, E); the resulting C_T is the free Σ-term category, and L_T is the quotient by E.

Remark 2.1 (Why Lawvere theories?). Lawvere theories are equivalent to finitary monads on **Set** [17]; thus every result herein can be re-phrased in the language of algebraic effects and handlers. We prefer Lawvere theories because they make the *syntactic* structure explicit: morphisms are term-tuples, composition is substitution, and the rewrite system arises directly from the presentation. In contrast, the monad formulation abstracts away the term structure, making it less suited to our goal of extracting *concrete* proof and execution certificates.

Models. A *model* of T is a product-preserving functor $M : T \to \mathbf{Set}$; notation $\mathrm{Mod}(T)$. For the syntactic theory presented by (Σ, E), a morphism $n \to 1$ in T is an equivalence class of n-variable Σ-terms modulo provable equality. The raw, unquotiented n-variable terms belong instead to the free term category C_T (equivalently, the free Σ-algebra before quotienting by E).

2.2 Equational logic

Given a signature Σ and a family E of equations, *equational logic* derives new equalities from E via four rules:

$$\text{refl},\ \text{sym},\ \text{trans},\ \text{congruence}.$$

Birkhoff's completeness theorem [4] says

$$E \vdash s = t \quad \Longleftrightarrow \quad (\forall M \in \mathrm{Mod}(T))\, M \models s = t.$$

Hence *proving* an equation and *holding in all models* are equivalent.

2.3 Term rewriting

Orienting each equation $l = r$ as a rule $l \to r$ yields a *rewrite system*. We define the *conversion relation* $\leftrightarrow^*$ as the reflexive, symmetric, transitive closure of one-step rewriting, closed under context and substitution. This is the computational counterpart to equational derivability: two terms are convertible iff they are provably equal.

Important distinction. In Section 4 we construct a *term category* C_T whose morphisms are raw term tuples and whose composition is substitution. The rewrite relation $\leftrightarrow^*$ is *not* a morphism in C_T; it is an external equivalence relation on the hom-sets. Rewrite steps relate parallel morphisms (same source and target), not composable ones. This separation—category structure vs. rewrite dynamics—is essential to the SLC formulation.

If the system is additionally *terminating* and *confluent*, the Church–Rosser property [9] yields a decision procedure: termination guarantees that every term has a normal form, and confluence guarantees uniqueness. Thus $s \leftrightarrow^* t$ iff $\mathsf{nf}(s) = \mathsf{nf}(t)$ syntactically—one computes both normal forms and compares. We emphasize that SLC does *not* require confluence; the correspondence holds for any Lawvere theory, but without confluence, path search replaces normalization.

Relation to rewriting logic. Meseguer's *rewriting logic* [23] provides an industrial-strength framework (realized in Maude [1]) where equations and labelled rewrite rules coexist. Our work differs in scope and purpose: we focus on the *unlabelled equational core* common to all Lawvere theories, emphasizing the categorical structure (L_T, C_T, projection P) and the *constructive bridge* between proof objects and rewrite traces. Rewriting logic is a powerful execution engine; SLC is a lightweight extraction framework for generating logic and computation from pure algebra.

Why SLC needs no confluence. The SLC framework relies only on the existence of rewrite paths, not on their uniqueness; thus, neither termination nor confluence is assumed. In practice, however, many theories (e.g. groups, rings) admit confluent completions via Knuth–Bendix [18], using termination orderings such as the lexicographic path ordering [12]. [1] For non-completable theories (e.g., monoids with associativity alone, where KB does not terminate), our tool falls back

[1] Practically, a non-confluent C_T means symbolic execution may branch and normal forms are not unique. Tool chains can still use C_T, but must explore (or bound) alternative paths instead of relying on canonical representatives.

Cheat-sheet.

- **Structure** T: operations & equations (syntax).
- **Logic** L_T: quotient category; provability = morphism equality.
- **Computation** C_T: raw term category; $\leftrightarrow^*$ is a relation on hom-sets.
- **SLC:** $s = t$ in all models $\iff P(s) = P(t)$ in $\mathsf{L}_T \iff s \leftrightarrow^* t$ in the rewrite graph over C_T.

Figure 2: Back-of-the-envelope guide to SLC.

to bounded bidirectional search; path-finding remains correct but may time out on long proofs. In our test suite, associativity-heavy proofs at depth 15–20 complete within seconds; deeper searches benefit from iterative deepening and best-first heuristics.

3 Running Example: Monoids

We instantiate the definitions above for the classical *monoid theory* T_{Mon}.

Signature and axioms

- Operation symbols: binary $*$ and constant e.
- Axioms: $(x * y) * z = x * (y * z)$ (assoc)
  $e * x = x = x * e$ (unit)

For rewriting we orient *both* sides of every axiom, so the rule set is $\{(x{*}y){*}z \leftrightarrow x{*}(y{*}z),\ e{*}x \leftrightarrow x,\ x{*}e \leftrightarrow x\}$.

Three viewpoints from the SLC framework

A. Structure T_{Mon}. Pure syntax of terms like $(x * e) * y$.

B. Logic L_T. Contexts n as objects; a morphism $n \to 1$ is an equivalence class of n-variable terms under provable equality. *Provability* is equality of morphisms in L_T; *proofs* (derivations witnessing equality) correspond to rewrite paths in the rewrite graph over the relevant hom-set of C_T.

C. Computation C_T. Objects are again the arities n. A morphism $n \to m$ is simply an m-tuple of *raw* terms $(t_1, \ldots, t_m)$ in the variables $x_1, \ldots, x_n$; no equations are imposed. On top of each hom-set we later add a *rewrite graph* generated by the bidirectional rules

$$(x * y) * z \ \leftrightarrow\ x * (y * z), \qquad e * x \ \leftrightarrow\ x, \qquad x * e \ \leftrightarrow\ x,$$

but those edges are *not* the morphisms—morphisms stay as raw tuples, while rewrites form an extra layer of structure used for computation.

Proof vs. execution

Logical derivation in L_T. Applying congruence with the unit axiom:

$$\frac{y * e = y \ \text{(unit)}}{x * (y * e) = x * y}\ \text{cong}$$

Rewrite path in the rewrite graph over C_T. Applying the *left-to-right* instance of the bidirectional rule $y * e \leftrightarrow y$ yields

$$x * (y * e) \xrightarrow{y*e \mapsto y} x * y.$$

The equational derivation witnesses equality in L_T, while the rewrite path lives in the rewrite graph on the relevant hom-set of C_T. They correspond under the SLC bridge and map to the same semantic equality in every monoid model, illustrating Theorem 4.1 on this concrete theory.

From proofs to paths. Each inference rule of equational logic corresponds to a simple manipulation in the rewrite graph over C_T: 'sym' reverses a path, 'trans' concatenates, and 'cong' lifts a step under context. Thus an entire proof tree in L_T unwinds, bottom-up, into a single composite path in the rewrite graph over C_T.

Normal forms

Restricting to the left-to-right orientation $(x * y) * z \to x * (y * z)$, $e * x \to x$, $x * e \to x$ yields a terminating, confluent system on monoid terms. Hence each term has a unique normal form: either e, or a right-associated product $x_1 * (x_2 * (\cdots * x_k) \cdots)$ with no occurrence of e. We use this as a sanity check in later sections.

This monoid toy will serve as an intuitive guide throughout the paper: each categorical construction in Sections 4–6 can be verified explicitly on T_{Mon}.

Example 2: Groups need symmetry

Let T_{Grp} extend T_{Mon} with a unary inverse operation $(\,\cdot\,)^{-1}$ and the extra axiom $x * x^{-1} = e$.

Why symmetry matters. SLC again yields the categories $\mathsf{L}_T^{\mathrm{Grp}}$, $\mathsf{C}_T^{\mathrm{Grp}}$. Because $\mathsf{C}_T^{\mathrm{Grp}}$ is built with the *bidirectional* rule

$$x * x^{-1} \;\leftrightarrow\; e,$$

an equational step in $\mathsf{L}_T^{\mathrm{Grp}}$ such as $e = x * x^{-1}$ (which uses the symmetry rule) is witnessed by traversing the rewrite edge $x * x^{-1} \rightsquigarrow e$ in the *reverse* direction, i.e. the conversion move $e \rightsquigarrow x * x^{-1}$ in the rewrite graph over $\mathsf{C}_T^{\mathrm{Grp}}$. If we had kept only the left-to-right orientation $x * x^{-1} \to e$, that reverse move would be impossible. Hence the two-sided generation of C_T is necessary whenever equational proofs rely on symmetry.

Example 3: Lattices and Boolean algebras

Our implementation includes richer algebraic structures. A *lattice* theory T_{Lat} has two binary operations (meet $\wedge$, join $\vee$) satisfying 8 axioms (associativity, commutativity, idempotence, and absorption laws). *Boolean algebra* T_{Bool} extends this with complement ($\neg$), identity elements (0, 1), and De Morgan laws, totalling 17 axioms.

These theories demonstrate that SLC scales beyond toy examples: each presentation yields its own L_T and C_T, with the inclusion morphism $T_{\mathrm{Lat}} \hookrightarrow T_{\mathrm{Bool}}$ transporting derivations and traces functorially.

4 The SLC Framework

We now state and prove the Structure–Logic–Computation framework. Fix an arbitrary (single-sorted) equational presentation (Σ, E). Let T denote the Lawvere theory it presents (the quotient category).

4.1 Statement

The SLC framework synthesizes two classical results into a unified categorical pipeline:

- **Birkhoff (1935):** provability $\iff$ validity in all models.
- **Term rewriting:** provability $\iff$ rewrite convertibility.

Formally, SLC packages these standard equivalences into a single constructive pipeline with checkable certificates and bidirectional derivation/trace translation.

Theorem 4.1 (Structure–Logic–Computation (SLC))**.** *Let (Σ, E) be an equational presentation, let L_T be the Lawvere theory it presents (provable equalities modulo E), and let C_T be the free term category on signature Σ (raw Σ-term tuples), with projection $P : \mathsf{C}_T \twoheadrightarrow \mathsf{L}_T$ quotienting by E. For any two parallel morphisms $f, g : n \to m$ in C_T (i.e., two m-tuples of raw Σ-terms), the following are equivalent:*

*(a) **(Logic)** Their images are equal in the logic category: $P(f) = P(g)$ in L_T.*

*(b) **(Computation)** They are connected by a rewrite path in the rewrite graph over C_T: $f \leftrightarrow^* g$.*

*(c) **(Semantics)** For every model $M : \mathsf{L}_T \to \mathbf{Set}$ (product-preserving functor), the interpretations agree: $M(P(f)) = M(P(g))$. Equivalently, for every set-based model and every valuation of the n variables, the two induced m-tuples evaluate equally.*

Scope of semantics. Throughout, a *model* of T means a product-preserving functor $M : \mathsf{L}_T \to \mathbf{Set}$ (equivalently $M : T \to \mathbf{Set}$, since $\mathsf{L}_T \cong T$ as the syntactic Lawvere theory presented by (Σ, E)). Thus item (c) quantifies over all set-based models of the theory. One could generalize to models in any category with finite products; we restrict to **Set** because (i) Birkhoff's completeness theorem is classically stated for set-based models, and (ii) our implementation targets concrete set-based interpretations.

Remark 4.2 (Multi-sorted and higher-order extensions). For brevity we state SLC in the single-sorted setting. For a k-sorted signature, the objects of both L_T and C_T become tuples in $\mathbb{N}^k$; the completeness argument adapts once Birkhoff's many-sorted theorem [14] is substituted for the single-sorted case. We have not implemented multi-sorted theories in our prototype; we state this extension as a standard result in the literature rather than a verified claim. Higher-order (cartesian-closed) theories require adding exponentials to the finite-product structure; the analogous correspondence is plausible but involves additional technical details (e.g., higher-order rewriting, $\beta\eta$-equivalence) that we do not address here.

Remark 4.3 (Smallness). For set-theoretic hygiene, all categories are assumed small (finitely generated from the signature). Readers comfortable with size issues may safely ignore this remark.

4.2 Construction and Proof

To prove Theorem 4.1, we first formally construct the categories L_T and C_T.

4.2.1 The Equational-Logic Category L_T

The equational category L_T (whose morphism equality captures provable equality) is the quotient of C_T by provable equality (i.e., the equational congruence generated by the axioms E), and is equivalent (as a category with finite products) to the syntactic Lawvere theory T presented by (Σ, E)—the equivalence is strict on objects and an isomorphism on hom-sets.

- **Objects:** Natural numbers $n \in \mathbb{N}$.
- **Morphisms:** A morphism $n \to m$ is an m-tuple $[t_1, \ldots, t_m]$ of Σ-terms in n variables, *modulo the equations* E.
- **Composition** is term substitution, which is well-defined as equality is a congruence. L_T is a cartesian category. **Functorial semantics.** A set-based model of T is a product-preserving functor $M : \mathsf{L}_T \to \mathbf{Set}$, which interprets each operation symbol as a function and hence each morphism as an m-tuple of functions. Composing with P yields an interpretation of raw terms in C_T as well.

Remark 4.4 (Morphisms vs. proof objects). Morphisms in L_T are *equivalence classes* of terms, not proof trees. To say $P(f) = P(g)$ is to assert *provable equality*; the morphism itself does not encode a derivation. When we speak of "proofs" in the implementation, we mean *derivation trees* (sequences of inference steps) that *witness* the equality—these are the `proof-step` structures in the code. The SLC bridge translates these derivation trees to traced rewrite paths and back.

4.2.2 The Term Category C_T

The term category C_T is the free cartesian category generated by the signature Σ. [2]

- **Objects:** Natural numbers $n \in \mathbb{N}$.
- **Morphisms:** An arrow $n \to m$ is an m-tuple of *raw* Σ*-terms* $(t_1, \ldots, t_m)$ in n variables. Syntactically distinct tuples are distinct morphisms.
- **Composition** is term substitution.

The logic category L_T is the quotient of C_T by the equational congruence $\equiv_E$ (provable equality from E), giving a canonical projection functor $P : \mathsf{C}_T \twoheadrightarrow \mathsf{L}_T$. Lemma 4.7 below shows that $\equiv_E$ coincides with the conversion relation $\leftrightarrow^*$, so $\mathsf{L}_T \cong \mathsf{C}_T/{\leftrightarrow^*}$.

Remark 4.5 (Category vs. rewrite relation). We emphasize that C_T is a category (raw term tuples as morphisms, composition by substitution), while the rewrite relation $\leftrightarrow^*$ is *external* structure on the hom-sets—a relation between parallel morphisms, not morphisms themselves. One could alternatively model computation via a *path category* $\mathsf{Path}(\mathsf{C}_T)$ whose morphisms are rewrite sequences; in that formulation, "computations are morphisms" becomes literally true. We prefer the simpler presentation where C_T is the syntax and the dynamics come from the rewrite graph defined *over* C_T.

Kernel = conversion: The SLC framework identifies the *kernel congruence* of the projection P (i.e., when $P(f) = P(g)$) with the conversion relation $f \leftrightarrow^* g$ (Lemma 4.7). Equivalently: L_T is the quotient of C_T by $\leftrightarrow^*$.

[2] That is, the *initial category with finite products* generated by the signature; see [22, §VI.5].

The Rewrite Relation. For computation, we define a *rewrite graph* G_T on each hom-set $\mathsf{C}_T(n, m)$: vertices are morphisms (term tuples), and there is an edge $f \rightsquigarrow g$ if g can be obtained from f by applying a single axiom of T (in either direction) to a subterm of one component. The conversion relation $f \leftrightarrow^* g$ is path-reachability in G_T—i.e., the reflexive-transitive closure of $\rightsquigarrow$. Computation in SLC means finding such paths; the path itself is the execution trace.

Remark 4.6 (Single terms vs. tuples). For clarity, we state the theorem for general m-tuples ($n \to m$). Our implementation focuses on the case $m = 1$ (single-term rewriting), which suffices for equational reasoning. Tuple convertibility is the product congruence generated componentwise: $(t_1, \ldots, t_m) \leftrightarrow^* (t'_1, \ldots, t'_m)$ iff each $t_i \leftrightarrow^* t'_i$. Rewrite paths may interleave component rewrites arbitrarily, but the induced equivalence relation factors through the product—each component's equivalence class is determined independently.

Lemma 4.7 (Kernel of projection = conversion)**.** *The kernel congruence of the projection functor $P : \mathsf{C}_T \twoheadrightarrow \mathsf{L}_T$ is precisely the conversion relation:*

$$P(f) = P(g) \text{ in } \mathsf{L}_T \quad \Longleftrightarrow \quad f \leftrightarrow^* g \text{ in } \mathsf{C}_T.$$

That is, L_T is the quotient of C_T by the conversion relation $\leftrightarrow^*$.

Proof. This follows from the standard fact that conversion (bidirectional rewriting) generates the same congruence as equational derivability [3, 5]. We prove both directions below via intermediate lemmas. □

Lemma 4.8 (Conversion is a congruence)**.** *The relation $\leftrightarrow^*$ on C_T-morphisms is:*

(i) an equivalence relation (reflexive, symmetric, transitive by definition);
(ii) closed under substitution: if $f \leftrightarrow^* g$ and σ is a substitution, then $\sigma(f) \leftrightarrow^* \sigma(g)$;
(iii) closed under context: if $f \leftrightarrow^* g$, then $C[f] \leftrightarrow^* C[g]$ for any context C;
(iv) containing the axioms: for each axiom $l = r$ of T and substitution σ, we have $\sigma(l) \leftrightarrow^* \sigma(r)$.

Proof. Properties (i)–(iii) follow from the definition of $\leftrightarrow^*$ as the reflexive-symmetric-transitive-context-substitution closure of one-step rewriting. Property (iv) holds because each axiom instance is a one-step rewrite. □

Lemma 4.9 (Conversion characterizes provability)**.** *The conversion relation $\leftrightarrow^*$ is the* least *congruence containing the axiom instances. Thus $f \leftrightarrow^* g$ if and only if $P(f) = P(g)$ in L_T (i.e., $f = g$ is derivable in equational logic from T). This is a standard result: both $\leftrightarrow^*$ and equational derivability define the same congruence closure of the axioms [3, 5].*

Proof. Equational derivability $\equiv_E$ is, by definition, the *least* congruence on Σ-terms containing the axiom instances. By Lemma 4.8, $\leftrightarrow^*$ is a congruence containing the axioms, so $\equiv_E \subseteq \leftrightarrow^*$. Conversely, every one-step rewrite $f \rightsquigarrow g$ is an axiom instance in context, hence $f \equiv_E g$; since $\equiv_E$ is closed under reflexivity, symmetry, and transitivity, it contains all of $\leftrightarrow^*$. Therefore $\leftrightarrow^* = \equiv_E$, and $f \leftrightarrow^* g$ iff $P(f) = P(g)$ in L_T. □

Lemma 4.10 (Soundness)**.** *If $f \leftrightarrow^* g$ in C_T then $P(f) = P(g)$ in L_T. Consequently, for every model $M : \mathsf{L}_T \to \mathbf{Set}$, we have $M(P(f)) = M(P(g))$.*

Proof. By Lemma 4.9, conversion equals provability. Since L_T is the quotient by provable equality, $P(f) = P(g)$. □

4.2.3 Proof of the SLC Theorem

The following lemma makes explicit the rule-by-rule translation that underlies the (a) $\iff$ (b) direction.

Lemma 4.11 (Inference-to-path translation)**.** *Each inference rule of equational logic corresponds to a path operation in the rewrite graph over* C_T*:*

Inference rule	***Path operation***	***Justification***
refl: $t = t$	*empty path at* t	*reflexivity of* $\leftrightarrow^*$
sym: from $s = t$ *to* $t = s$	*reverse path*	*symmetry of* $\leftrightarrow^*$
trans: from $r = s$, $s = t$ *to* $r = t$	*concatenate paths*	*transitivity of* $\leftrightarrow^*$
cong: from $s = t$ *to* $C[s] = C[t]$	*lift each step into* $C[-]$	*closure under context*
axiom: $l = r$ *(instance)*	*single* $\rightsquigarrow$ *step*	*axiom generates edge*

Conversely, each single rewrite step $f \rightsquigarrow g$ *is an axiom instance in context, hence provable by* axiom + cong*; a path is provable by* trans*-chaining its steps.*

Proof. Immediate from the definition of $\leftrightarrow^*$ as the reflexive-symmetric-transitive-context closure of axiom instances. □

Proof of Theorem 4.1. We establish the equivalence of (a), (b), and (c) for parallel morphisms $f, g : n \to m$ in C_T. The individual implications are classical (modulo standard completeness theorems); our contribution is their categorical packaging and the explicit translation of Lemma 4.11.
(a) $\iff$ (c): (Logic $\iff$ Semantics) This is Birkhoff's completeness theorem [4] (many-sorted: [14]), rephrased in Lawvere's categorical language [20]. An equality is provable in the theory (i.e., $P(f) = P(g)$ in L_T) if and only if it holds in all models.
(a) $\iff$ (b): (Logic $\iff$ Computation) This is the standard identification of equational derivability with the congruence generated by contextual axiom instances [3]; we include it to make the bridge explicit and implementable.

($\Rightarrow$): By structural induction on the proof tree, applying Lemma 4.11 at each node. Each inference rule maps to a path operation; composing these yields $f \leftrightarrow^* g$.

($\Leftarrow$): Each rewrite step $f_i \rightsquigarrow f_{i+1}$ is an axiom instance in context, hence provable by axiom + cong. Chain the steps by trans to obtain $P(f) = P(g)$.

The novelty lies not in these implications but in making them *constructive*: our implementation (Section 5) translates proofs to traced certificates and vice versa, following the correspondence of Lemma 4.11. □

5 An SLC Implementation in Racket

To demonstrate feasibility and provide an auditable reference implementation, we have developed a prototype system in Racket [8] that directly implements the structure-to-logic/computation pipeline. The system takes an equational presentation (Σ, E) as input and provides an interactive environment for both proving theorems in L_T and simulating execution paths in the rewrite graph over C_T.

Implementation status. Table 2 summarizes what is implemented versus planned. The core pipeline is complete: theory definition, equality checking via path search, multiple search strategies, traced certificates with full step metadata, bidirectional proof/trace translation, and theory morphisms with bounded verification. Knuth–Bendix completion support is partial and not yet production-ready.

Decidability caveat. Equality in arbitrary Lawvere theories is undecidable in general (it subsumes the word problem for finitely presented algebras). Our tool is therefore a *semi-decision procedure*: given a depth bound, it searches for a rewrite path; if one exists within the bound, the tool finds it and returns a certificate. If no path is found, this means either the terms are not equal or the bound was too small. For confluent, terminating theories (detected via optional Knuth–Bendix completion), the tool normalizes both terms and compares syntactically, which is a decision procedure.

Search-space management. Bidirectional rewriting can exhibit exponential branching in the worst case. We mitigate this via (i) iterative deepening with configurable depth bounds, (ii) best-first search with term-size heuristics, and (iii) optional KB pre-simplification to reduce term complexity before search. For the theories in our test suite (monoid, group, lattice, Boolean algebra), proofs of typical identities complete in milliseconds at depths 10–20; pathological cases requiring longer paths are handled by increasing bounds or applying completion when available.

Feature	**Status**	**Notes**
Theory definition (signature, axioms)	Implemented	Core DSL
Equality checking (path search)	Implemented	BFS, IDS, best-first
Multiple example theories	Implemented	Monoid, Group, Lattice, Boolean, Ring
Traced rewrite steps	Implemented	Axiom id, direction, subterm path, bindings
Bidirectional LT↔CT translation	Implemented	proof→trace and trace→proof
Theory morphisms	Implemented	Definition, application, bounded verification
Knuth–Bendix completion	Partial	LPO, critical pairs; completion loop incomplete
Certified export (Lean/Coq)	Planned	Not yet implemented

Table 2: Implementation status of claimed features.

Trace certificate structure. Each rewrite step is recorded as a `rewrite-step` struct (in the `ct.rkt` module) containing: axiom index, orientation (forward/backward), subterm path, and variable bindings. A complete rewrite path is a `traced-path` struct assembling the sequence of steps with start/end terms and theory name. These certificates are designed for independent auditing: given the theory and a traced path, a third-party checker could mechanically verify each step.

5.1 Defining Structure

A user provides the structure—an equational presentation (Σ, E)—by defining its signature and axioms. The operations are defined implicitly by their use in the axioms. For our running example, the monoid theory T_{Mon} is defined as follows (from `slc/examples/monoid.rkt`):

```
(define T_Mon
  (lawvere-theory
```

```
  'Monoid
  (list
   ;; Associativity: (x*y)*z = x*(y*z)
   (cons (m* (m* (mvar 'x) (mvar 'y)) (mvar 'z))
         (m* (mvar 'x) (m* (mvar 'y) (mvar 'z))))
   ;; Left identity: e*x = x
   (cons (m* me (mvar 'x)) (mvar 'x))
   ;; Right identity: x*e = x
   (cons (m* (mvar 'x) me) (mvar 'x)))))
```

Listing 1: Defining the monoid theory in Racket.

5.2 Logic as Proof, Computation as Simulation

The system's core function, `prover+simulator`, operationalizes the SLC framework. To *attempt* a proof of equality in L_T, the system searches for a rewrite path in the rewrite graph over C_T. When a path is found, it serves as a checkable certificate of equality. If no path is found within the search bound, the result is inconclusive (absence of a witness within the bound is not a disproof).

Implementation note. Mathematically, L_T is defined as the quotient of C_T by provable equality—each equivalence class is a morphism in L_T. Our implementation does *not* construct these equivalence classes explicitly (which would be infinite for most theories). Instead, we represent L_T-equality *operationally*: two terms are equal in L_T iff a rewrite path connects them in the rewrite graph over C_T. Theorem 4.1 establishes soundness and completeness *mathematically*; operationally, our bounded search is sound and complete *up to the chosen depth bound*, yielding a checkable certificate when it succeeds.

We can invoke the system from an interactive REPL. For example, to prove that $(x*e)*y = x*y$ in the monoid theory, a user would type:

```
slc> use-theory Monoid
slc> prove (x*e)*y = x*y
```

The system confirms the proof and provides the computational trace that justifies it, demonstrating the equivalence between L_T-provability and C_T-convertibility:

```
 Proved: ((x * e) * y) = (x * y)
Proof path:
  Step 0: ((x * e) * y)
  Step 1: (x * y)
```

Decidability and soundness. The SLC theorem establishes *mathematical equivalence*: path-reachability in the rewrite graph over C_T equals provability in L_T. Our *algorithmic implementation* performs bounded depth-limited search and is a semi-decision procedure: any returned path is a *sound* certificate of equality, but failure to find a path within the bound is inconclusive (the terms may still be equal, but require a longer derivation). For confluent terminating theories (detectable via Knuth–Bendix completion), normalization provides a decision procedure.

The boolean result ('Proved') is the answer in the logic category L_T. The explicit path is the witness from the rewrite graph over C_T. The prototype includes multiple pathfinding strategies (BFS, iterative deepening, and best-first search with term-size heuristics); by default, it selects automatically based on problem characteristics. Performance optimizations like caching make it practical for exploring the consequences of algebraic theories.

Tool guarantees. The mathematical correspondence says: provable equality $\iff$ existence of a conversion path. The tool, however, only *searches* for paths, so:

- **Sound:** When the tool returns a trace, it is a valid certificate, structured for independent checking.
- **Incomplete:** Failure to find a trace within the search bound is *not* evidence of disequality—the path may exist beyond the bound.
- **Decision procedure:** For theories admitting Knuth–Bendix completion, normalization provides a true decision procedure (compare normal forms).

5.3 Traced computation and proof certificates

A central engineering lesson is that a rewrite path is only useful as a *certificate* if each step records *why* it was legal—echoing the design of proof-carrying code [26]. Our initial path-finder returned only a sequence of terms; this loses the axiom instance, the subterm rewritten, and the substitution used, making proof reconstruction impossible.

We therefore implement *traced rewriting*. Each transition records (i) the axiom index and orientation, (ii) a subterm path into the term tree, and (iii) the variable bindings (matching substitution). From such a trace we reconstruct a formal equational proof by instantiating each axiom, optionally applying symmetry, lifting the equality into context via congruence, and chaining steps by transitivity. The result is a proof object in L_T together with a human-readable explanation.

Step correctness. Each trace step is validated at construction time: (i) the substitution is computed by standard first-order pattern matching, and we verify that applying it to the axiom's LHS yields the matched subterm exactly; (ii) the context path is checked to be a valid index sequence into the term tree; (iii) the result term is computed by substituting the axiom's RHS (under the same matching substitution) back into the context. If any check fails, the step is rejected. This ensures that each individual step is locally correct; the full trace is correct by induction:

```
slc> show-proof (e * X) * e = X
1. [axiom]  (e * X) = X
2. [cong]   ((e * X) * e) = (X * e)
3. [axiom]  (X * e) = X
4. [trans]  ((e * X) * e) = X
```

Certificate payload. The above is the human-readable rendering. The underlying `traced-path` structure contains the full auditable metadata for each step. For example, step 1 (applying the left-identity axiom $e * x = x$ to the subterm $(e * X)$) is recorded as:

```
rewrite-step {
  from-term:       (e * X)
  to-term:         X
  axiom-index:     1           ; left-identity axiom
  axiom-direction: forward     ; l->r orientation
  subterm-path:    []          ; rewrite at root position
  bindings:        {x |-> X}   ; variable substitution
}
```

This payload enables *independent verification*: given the theory's axioms and a `traced-path`, a third-party checker could mechanically validate each step without trusting the path-finding algorithm. The certificate is self-contained: it includes the axiom index (pointer into the theory), the direction (left-to-right or right-to-left), the exact subterm position (path into the term tree), and the variable bindings (matching substitution).

Conversely, we implement the reverse translation: given a proof term, we extract a traced rewrite sequence by mapping each inference rule to its computational counterpart (axiom $\mapsto$ one rewrite, symmetry $\mapsto$ direction reversal, transitivity $\mapsto$ concatenation, congruence $\mapsto$ contextual rewriting).

Bidirectional translation contract. The SLC bridge guarantees bidirectional totality:

- `proof`→`trace` is *total*: every equational derivation (proof object in L_T) converts to a traced rewrite path in the rewrite graph over C_T.
- `trace`→`proof` is *total*: every valid traced path (sequence of `rewrite-step`s) converts to a formal equational derivation.
- The prototype includes partial certificate-validation support (`verify-proof`). Proof validation currently checks structural well-formedness and direct axiom instances; a dedicated trace checker and a full small trusted kernel are planned for future work.

This separation of *generation* (path search, proof construction) from *verification* (certificate checking) follows the proof-carrying code paradigm. The current validator is lightweight; strengthening it into a small, audited checker that independently validates certificates without trusting the generator is a priority.

Round-trip invariant. The bidirectional translation satisfies an engineering invariant validated by our test suite:

- `trace`→`proof`→`trace` returns a trace with the same start/end terms;
- `proof`→`trace`→`proof` returns a proof of the same equation, equivalent up to normalization.

Scope of validation: We do not claim a formally verified meta-theorem (e.g., in Coq or Lean). Rather, these properties are checked by unit tests on hand-crafted and randomly generated examples across our running theories (Monoid, Group, Lattice, Boolean). The test suite covers: (i) all axiom instances, (ii) composition of 2–5 steps, and (iii) proofs involving each inference rule (sym, trans, cong). This empirical validation serves as a practical correctness check but is not a substitute for machine-checked formalization, which we leave to future work (Section 8).

Theory	**Example**	**Path len.**	**Proof size**	**Time (ms)**
Monoid	$(e * x) * e = x$	2	4	0.3
Lattice	$x \wedge (x \vee y) = x$	1	2	0.1
Boolean	$x \wedge \neg x = 0$	1	2	0.2

Table 3: Certificate sizes for representative equalities. Path length counts rewrite steps; proof size counts inference nodes.

5.4 Structure as functoriality: theory morphisms

To make the "Structure" vertex operational, we implement morphisms of Lawvere theories. Categorically, a *theory morphism* $\varphi : T_1 \to T_2$ is a finite-product-preserving functor that is the identity on objects. Operationally, we represent φ by interpreting each operation symbol f of T_1 as a term $\varphi(f)$ in T_2 (of the same arity), and verifying that for every axiom $l = r$ of T_1, the translated equation $\varphi(l) = \varphi(r)$ is provable in T_2. Such a morphism induces functors φ_* on both the term

and logic categories by substitution/translation, making the following diagram commute:

$$\begin{array}{ccc} \mathsf{C}_{T_1} & \xrightarrow{P_1} & \mathsf{L}_{T_1} \\ {\scriptstyle \varphi_*}\downarrow & & \downarrow{\scriptstyle \varphi_*} \\ \mathsf{C}_{T_2} & \xrightarrow[P_2]{} & \mathsf{L}_{T_2} \end{array}$$

Concretely, φ_* transports terms, proofs, and traced paths. Commutativity of the diagram is demonstrated operationally: we check that the transported proof conclusions and path endpoints agree, rather than proving full categorical equality of proof objects. Our verifier checks axiom preservation via bounded search (depth 10 by default) and is therefore sound when it accepts but incomplete (it may fail to verify a valid morphism if the required path exceeds the search depth). This is the main engineering limitation of the current morphism support; replacing bounded search with certificate-based checks (e.g., via KB completion) is a priority for future work. As small case studies we include the inclusion of lattice theory into Boolean algebra (8 axioms $\hookrightarrow$ 17 axioms), and the embedding of monoids into groups.

5.5 Abstract machine interpretation

The rewrite perspective admits an operational reading: the rewrite graph layered over C_T defines an abstract machine whose configurations are terms, transitions are rewrite steps, and traces are execution histories. We implement a simple abstract machine that makes this correspondence explicit:

- A *configuration* is a term together with its accumulated trace;
- A *transition* applies one axiom (in either direction) at some subterm, recording the step;
- *Execution* runs until a target term is reached or no more steps apply.

In bidirectional rewriting, "normal form" is not unique—variables can always expand (e.g., $x \to e * x$). The machine-trace view connects rewriting to operational semantics: confluence corresponds to determinism, and a traced path *is* an execution certificate.

5.6 Completion, normalization, and empirical complexity

Finally, we explore when the computational side admits fast equality checking via normalization. We implement preliminary Knuth–Bendix support (critical-pair generation, LPO-based orientation, partial completion) and perform exploratory benchmarks on (i) theories that complete quickly (idempotence, unital laws), and (ii) theories that do not (monoids with associativity). For fully completable theories (e.g., idempotent: $x * x = x$), normalization can yield orders-of-magnitude speedups in small benchmarks, because equality reduces to syntactic comparison of normal forms. For theories with associativity, KB produces only partial completion; speedups are modest and sometimes negative due to normalization overhead. We use KB completion as an *optimization when it succeeds*, not as a requirement.

We also implement proof normalization (eliminating redundant uses of symmetry/transitivity via rules like $\mathsf{sym}(\mathsf{sym}(p)) \to p$) and instrument both translations (trace→proof and proof→trace) with size and time metrics. The current implementation performs certificate translations in time roughly proportional to the number of trace/proof steps; exact asymptotics depend on term representation and sharing.

Reproducibility. The reference implementation is available on GitHub [8]. To reproduce the examples: `racket slc/interactive.rkt` launches the REPL; `raco test tests/` runs the core test suite (terms, theories, CT, LT, and simulator modules). Benchmarks in `slc/benchmarks.rkt` output timing data to stdout. Traced certificates are returned as Racket data structures; a JSON export is planned but not yet implemented.

6 Significance

The SLC framework treats algebraic structure as a unifying layer connecting logic (proofs) and computation (programs).

- **Domain-specific languages (DSLs).** This is the most immediate application. A developer defines the *business logic* of a domain as an equational presentation (Σ, E)—operations and their equational laws—and the SLC pipeline automatically provides (i) an equational prover for domain identities and (ii) a symbolic executor for domain expressions. No ad-hoc proof system or operational semantics is required; both are derived systematically from the structure.

- **Language design.** Given a new effect signature (say $\mathsf{push}/\mathsf{pop}$), systematically derive (i) an equational reasoning layer (L_T) and (ii) a rewrite semantics whose traces live in the rewrite graph over C_T. Soundness arguments reduce to standard algebraic reasoning once the structure is fixed.

  Tiny illustration. Consider a simple "stack" signature with push, pop, and axiom $\mathsf{pop}(\mathsf{push}(x, s)) = s$. SLC yields (i) a logic proving stack-manipulation identities and (ii) a rewrite semantics that symbolically evaluates stack expressions. (Richer effect signatures like exceptions require enriched or indexed Lawvere theories [31], beyond our scope.)

- **Formal verification (potential).** If a compilation pipeline is itself modeled as a verified theory morphism preserving the relevant equations, then SLC suggests a route to transporting equalities and trace witnesses across compilation stages. Significant work would be required: semantics-preservation properties, handling of effects, and integration with existing compiler infrastructure are all open challenges. Solver-aided DSLs such as Rosette [33], symbolic executors like KLEE [6], and auto-active verifiers (e.g. Dafny [21]) all follow a structure→logic→execution workflow, providing concrete evidence that the SLC pipeline pattern scales to practical tool chains.

- **Security modelling (speculative).** In lattice-based information-flow models [11], security levels form a lattice. Privilege-escalation conditions could be modeled as reachability queries in a rewrite system induced by the policy algebra: can a low-privilege term reach a high-privilege form? A discovered path serves as a concrete attack witness; absence of a path within the search bound provides bounded assurance. This application is speculative—real security proofs require threat modeling, composition guarantees, and side-channel analysis beyond our scope. Stateful or effectful security properties would require enriched Lawvere theories [31].

Bidirectional vs. directed rewriting. We use bidirectional rewriting ($\leftrightarrow^*$) because equational logic is inherently symmetric. In traditional computer science, "computation" often implies *directed* reduction ($\rightarrow$) toward a normal form. When a theory is confluent and terminating, directed rewriting suffices and yields efficient decision procedures (normalize both sides, compare). For

general Lawvere theories—which need not be confluent—bidirectional rewriting is the correct notion: it captures *equivalence checking* rather than *evaluation*. Our framework supports both: when Knuth–Bendix completion succeeds, we use normalization; otherwise, we fall back to path search.

7 Related Work

Rewriting logic and Maude. Meseguer's rewriting logic [23] treats equations as labelled rewrite rules modulo congruence; our C_T corresponds to its unlabelled equational core, with L_T providing the associated proof calculus. A twenty-year retrospective appears in [24], and the Maude system [1] demonstrates industrial-scale tooling based on the same equation + rewrite pattern. Our focus is different: rather than a full rewriting-logic metalanguage, we emphasize a small certificate format and bidirectional derivation↔trace translation.

Initial-algebra and structural operational semantics. The algebraic-specification tradition uses syntax-generated algebraic structure to define denotational meaning, while Plotkin's structural operational semantics [30] gives rule-based operational meaning. Rewriting logic is especially relevant because it explicitly unifies algebraic denotational semantics and SOS within a single framework [24]. SLC packages the equational core of this pattern into a generic algebra ⇒ logic + computation pipeline.

Algebraic effects and handlers. Algebraic effects [29] begin from operations and equations; their free models induce the corresponding computational monads. Handlers add further structure on top of that base theory. SLC mirrors the "algebra → equations → computation" pattern at the equational core, providing a completeness guarantee for the first-order equational fragment: the equational laws of effects correspond to convertibility in the rewrite graph over C_T.

Congruence closure and equality saturation. The congruence closure algorithm [27] efficiently decides ground equality in equational theories—a key subroutine in SMT solvers. Equality saturation via e-graphs [36] extends this idea to term optimization: saturate an equivalence class with all provably equal terms, then extract an optimal representative. Our work is complementary: we focus on *certificates* (traced paths) rather than decision procedures, and on *open terms* (with variables) rather than ground terms. E-graph techniques could accelerate our path search when applicable.

Proof-carrying code. Necula's proof-carrying code [26] and its semantic foundations [2] demonstrate the value of attaching verifiable certificates to executables. Our traced rewrite certificates serve a similar role at the equational level: a path in the rewrite graph over C_T is a "proof" that can be checked independently of how it was found.

8 Future Work

We outline some directions that naturally extend the present study.

- **Certified export.** Emit proof objects to external checkers (Lean/Coq) or interoperable formats (TPTP), and ship a small, audited kernel that re-checks certificates produced by the tool.

- **Completion beyond first-order LPO.** Extend the completion engine with better orderings (KBO, polynomial orders) [12] and rewriting modulo associativity/commutativity (AC unification and AC completion) [28, 13], which is essential for realistic algebraic theories.
- **Scaling and indexing.** Improve rewriting/search performance via term indexing (discrimination trees, substitution trees) [32], memoization of normal forms, and parallel critical-pair generation.
- **Stronger morphism verification.** Replace bounded search in `verify-morphism` with certificate-based checks (completion when available, or external equational provers) and extend morphisms to multi-sorted/typed theories.
- **Larger case studies.** Evaluate the framework on richer theories (groups, rings, modules) and on domain-specific equational presentations (e.g., instruction semantics fragments, protocol algebras), where certificates are practically valuable.
- **Visualization and audit logs.** Provide interactive views of proof trees and rewrite traces, and a structured "audit log" format for downstream tooling.

9 Conclusion

We have presented the Structure–Logic–Computation (SLC) framework, a constructive operationalization of the classical correspondence between equational presentations, Lawvere theories, and term rewriting. Given an equational presentation (Σ, E), SLC derives two canonical categories—the logic category L_T (provable equalities) and the term category C_T (raw term tuples, equipped with a rewrite graph)—with a bidirectional translation between equational derivations and traced rewrite certificates.

The mathematical foundations (Birkhoff completeness, conversion = provability) are well established; our contribution is making these equivalences *executable and auditable*. The Racket prototype demonstrates feasibility: equational proofs convert to witnessed rewrite paths and vice versa, theory morphisms transport both structures functorially (with commutativity demonstrated operationally in the prototype), and Knuth–Bendix completion (when applicable) provides decision procedures via normalization.

Limitations. Equality in arbitrary presentations is undecidable; our tool is a semi-decision procedure with bounded search. For non-confluent theories without terminating completions, the search may fail to find proofs within resource bounds even when they exist. The prototype handles single-sorted, first-order theories; multi-sorted and higher-order extensions require additional infrastructure.

Impact. SLC provides a rigorous bridge between algebraic specifications and verified computation. The framework enables: (i) domain-specific languages with built-in equational reasoning, (ii) auditable proof certificates for security-critical transformations, (iii) functorial transport of verified properties across theory morphisms, and (iv) a principled foundation for tools that manipulate algebraic structures.

The reference implementation [8] accompanies this paper, inviting extension and evaluation on richer theories and practical case studies.

References

[1] The Maude system. https://maude.cs.uiuc.edu/, 2025. URL (accessed May 12, 2025).

[2] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, pages 243–253. ACM, 2000. doi:10.1145/325694.325727.

[3] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.

[4] Garrett Birkhoff. On the structure of abstract algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, 1935. doi:10.1017/S0305004100013463.

[5] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra*, volume 78 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1981. Millennium Edition (2012) freely available online.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, San Diego, CA, 2008. USENIX Association.

[7] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010 – Concurrency Theory (21st Int'l Conference)*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

[8] Martila Research. A Racket Implementation of the Structure-Logic-Computation (SLC) Framework, 2025. https://github.com/Martila-iO/slc.

[9] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936. doi:10.1090/S0002-9947-1936-1501858-0.

[10] Ranald Clouston. Nominal Lawvere Theories: A category theoretic account of equational theories with names. *Journal of Computer and System Sciences*, 80(6):1067–1086, 2014. doi:10.1016/j.jcss.2014.04.003.

[11] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. doi:10.1145/360051.360056.

[12] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982. doi:10.1016/0304-3975(82)90026-3.

[13] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. *Handbook of Theoretical Computer Science*, B: Formal Models and Semantics:243–320, 1990. Comprehensive survey of term rewriting including completion and AC.

[14] Joseph A. Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[15] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Originally circulated as manuscript in 1969.

[16] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, 2007. Modern treatment of Lawvere theories and their equivalence to finitary monads. doi:10.1016/j.entcs.2007.02.019.

[17] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, 2007. doi:10.1016/j.entcs.2007.02.019.

[18] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[19] Joachim Lambek. Deductive systems and categories II: Standard constructions and closed categories. In Peter J. Hilton, editor, *Category Theory, Homology Theory and their Applications I*, volume 86 of *Lecture Notes in Mathematics*, pages 76–122. Springer, Berlin, 1969. doi:10.1007/BFb0079385.

[20] F. William Lawvere. *Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963. Ph.D. thesis; reprinted in *Reprints in Theory and Applications of Categories* No. 5 (2004), pp. 1–121.

[21] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

[22] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1971. 2nd ed. published in 1998.

[23] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. doi:10.1016/0304-3975(92)90136-2.

[24] José Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81(7–8):721–781, 2012. doi:10.1016/j.jlap.2012.04.006.

[25] Rasmus Ejlers Møgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer Science*, 10(1:17), 2014. doi:10.2168/LMCS-10(1:17)2014.

[26] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119. ACM, 1997. doi:10.1145/263699.263712.

[27] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980. doi:10.1145/322186.322198.

[28] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981. Foundational work on AC-completion. doi:10.1145/322248.322251.

[29] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

[30] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI FN-19, Computer Science Dept., Aarhus University, 1981. Revised version (with corrections) published 2004.

[31] John Power. Indexed Lawvere theories for local state. In B. Hart, T. G. Kučera, A. Pillay, P. J. Scott, and R. A. G. Seely, editors, *Models, Logics and Higher-Dimensional Categories: A Tribute to the Work of Mihály Makkai*, volume 53 of *CRM Proceedings & Lecture Notes*, pages 213–229. American Mathematical Society, Providence, RI, 2011.

[32] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1964. Elsevier, 2001. doi:10.1016/B978-044450813-3/50028-X.

[33] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*, pages 135–152, Indianapolis, IN, 2013. ACM. doi:10.1145/2509578.2509586.

[34] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2–3):384–418, 2014. Special issue for ICFP 2012. doi:10.1017/S095679681400001X.

[35] Philip Wadler. Propositions as Types. *Communications of the ACM*, 58(12):75–84, December 2015. doi:10.1145/2699407.

[36] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. In *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2021)*, pages 1–29. ACM, 2021. doi:10.1145/3434304.