# SODA: A Language for Executable Semantics

William Kirkman-Moller

https://github.com/soda-lang-project/SODA

**Abstract**

SODA is a domain-specific language for writing, running and comparing operational, denotational and axiomatic semantics. Language designers define syntax once and use it for execution rules, semantic functions and proofs. SODA evaluates these definitions, displays derivations and checks proof certificates. We demonstrate their agreement for a language with assignments, conditionals and local variables. Lean proofs establish operational–denotational correspondence, soundness and relative completeness of its Hoare rules, and exact symbolic weakest preconditions. A checker validates results exported by SODA against this model. An account-transfer program applies the theorems to prove conservation of balances across arbitrary finite batches.

## 1 Introduction

A programming language may have several semantic descriptions. Execution rules explain how a program runs, a denotational interpretation assigns it a mathematical meaning, and proof rules justify assertions about its behaviour. Each description must account for the same syntax and variables, even when developed separately. SODA lets language designers write all three in one module and execute them over the same terms.

SODA supplies shared term and substitution operations for these definitions. It evaluates terms, constructs derivations, checks proof certificates and compares interpretations on supplied inputs or finite domains. This follows the tradition of executable semantic models in Redex and K, and of support for syntax and binding in Ott [1, 3, 2]. Section 2 shows how execution, interpretation and proof share one arithmetic syntax, including an extension with a new construct.

We developed SODA to investigate semantic choices in X-Lang, a systems language with explicit optimisation declarations. X-Lang permits local mutation in pure functions and generates LLVM code. We use SODA to examine how updates survive local scope exit and which integer transformations preserve behaviour in the presence of overflow and undefined values. Both studies are written as executable semantic definitions.

For the formal study, we use a language with assignment, sequencing, conditionals and local scopes. Lean proves agreement between its execution rules, state transformers and Hoare reasoning, including exact symbolic weakest preconditions. A result checker connects these theorems to computations exported by SODA. Table 2 identifies the role of each component. An account-transfer program applies the correspondence to conservation of balances across arbitrary finite batches.

# 2 SODA by example

## 2.1 Defining and running semantics

Listing 1 defines integer literals and addition, then gives both an evaluation judgment and a compositional interpretation. Both use the same `Expr` constructors. An `infer` clause states a conclusion and its premises; a `rule` gives an executable equation for a function. Metavariables begin with `?`.

Listing 1: Two executable accounts of the same arithmetic language.

```
module Arithmetic;
sort Expr;
constructor lit(Int): Expr;
constructor plus(Expr, Expr): Expr;
judgment evaluates(Expr, Int) kind evaluation;

infer literal: evaluates(lit(?n), ?n);
infer addition: evaluates(plus(?a, ?b), ?result)
    if evaluates(?a, ?x), evaluates(?b, ?y)
    let ?result = add(?x, ?y);

function meaning(Expr): Int;
rule meaning_literal: meaning(lit(?n)) => ?n;
rule meaning_plus: meaning(plus(?a, ?b))
    => add(meaning(?a), meaning(?b));

query evaluates(plus(lit(2), lit(3)), ?value);
eval meaning(plus(lit(2), lit(3)));
```

The query returns `?value = 5` with a derivation containing two literal rules and one addition rule. Its computed `let` premise adds the values found by the recursive premises. Running `meaning` by rewriting or compositional evaluation also returns 5. Both operations take the same expression as input.

## 2.2 Extending the same syntax

Suppose the language gains `twice(e)`, which doubles an expression's value. Listing 2 adds one constructor and a case for each semantic account. The existing literal and addition cases, the judgment signature and the `meaning` signature remain unchanged.

Listing 2: Adding one construct to execution and interpretation.

```
constructor twice(Expr): Expr;
infer doubling: evaluates(twice(?e), ?result)
    if evaluates(?e, ?n) let ?result = add(?n, ?n);
rule meaning_twice: meaning(twice(?e))
    => add(meaning(?e), meaning(?e));
```

The same terms also support proof checking. We supply the defining equations for `meaning_twice` and `meaning_plus` as explicit axioms, separately from their execution rules. A two-step certificate unfolds the first equation and uses the second in reverse. Writing $M$ for `meaning`, it checks

$$\begin{aligned} M(\text{twice}(e)) &= \text{add}(M(e), M(e)) \\ &= M(\text{plus}(e, e)). \end{aligned}$$

Inference, evaluation and certificate checking use the same constructors and substitution operations, so the extension requires no conversion between term representations. The author supplies each new semantic case and the equations used in the proof; SODA checks their sorts and the certificate's equation applications.

For $e = \mathrm{plus}(\mathrm{lit}(2), \mathrm{lit}(3))$, inference returns 10 for both `twice(e)` and `plus(e,e)`, with checked derivations. Rewriting and compositional evaluation of their meanings also return 10. The original expression still gives 5 under inference and both evaluators, and the equational certificate passes the proof checker.

The next example adds mutable state and local variables, comparing execution with a state transformer and a symbolic precondition calculation. Its proof rules and correspondence theorems are developed in Section 3.

### 2.3 Following state through a local scope

Consider a program that saves a balance in a temporary variable and then updates the balance. In conventional program notation, the command is

Listing 3: A temporary snapshot whose outer update survives scope exit.

```
{
    let snapshot = balance;
    balance = snapshot + 1;
}
```

The temporary snapshot disappears on exit; the balance must retain its new value. Restoring the entire incoming store would lose the update. We use this program to check how execution, denotation and precondition calculation handle scope exit.

The module `state-correspondence.soda` defines this language using indexed variables. The balance occupies slot zero initially and slot one inside the local scope, where the new slot zero holds the snapshot. Let $c$ be the command above, let $s = [2]$ be its initial store, and let $Q$ assert that the final balance equals three. Numerals and list brackets abbreviate the structural constructors described in Section 3. Table 1 shows the results from the module's `executes` judgment, `meaning` function and symbolic transformer `weakest`.

| Operation | Result |
|---|---|
| `executes(c,s,?out)` | `?out` $= [3]$, with a checked derivation. |
| `meaning(c,s)` | $[3]$ under both rewriting and compositional evaluation. |
| `weakest(c,Q)` | $\mathrm{var}(0) + 1 = 3$ under both evaluators. |
| Independent Lean check | Both exported results accepted. |

Table 1: Three accounts of Listing 3. The calculated precondition describes the initial balance needed to obtain a final balance of three.

Both the final store and the calculated precondition retain the outer balance update and discard the temporary variable. The precondition holds for the initial balance of two. The Lean checker accepts the exported final stores and preconditions for the expected command, input and assertion. Section 3 proves the correspondence for every core program; Section 3.6 gives the acceptance theorem.

| Component | Role and evidence |
|---|---|
| SODA toolkit | Runs user-defined syntax, rules and functions; checks derivations and proof certificates; compares interpretations on chosen inputs or finite domains. |
| Stateful core | A particular language. Lean proves semantic correspondence, Hoare soundness and relative completeness, and exact weakest preconditions. |
| Result checker | Validates computations exported for that core. Acceptance establishes the final store and precondition for the caller's expected command, input and assertion. |

Table 2: The toolkit, the formal study and the connection between them.

# 3 Correspondence for a stateful core

The stateful language has assignment, sequencing, a zero-test conditional and local scope:

$$e ::= n \mid \mathrm{var}(i) \mid e + e \mid e \mathbin{\dot{-}} e,$$
$$c ::= \mathrm{skip} \mid i := e \mid c; c \mid \mathrm{ifzero}(e, c, c) \mid \mathrm{local}(e, c).$$

Numbers and indices are natural numbers, and $a \mathbin{\dot{-}} b = \max(a - b, 0)$ is truncated subtraction. A store is a finite list with an implicit zero-filled tail; writing beyond its end allocates the intervening zero entries. SODA represents numbers and indices with `zero` and `next` constructors; its structural arithmetic uses no host integer operations.

A local scope evaluates its initialiser in the incoming store, pushes that value at index zero, runs its body, and discards the local slot. Outer indices shift by one during the body, and updates to those locations survive scope exit.

## 3.1 Operational and denotational agreement

Write $E(e, s)$ for expression evaluation, $s[i \mapsto v]$ for store update, and $\mathcal{D}(c)$ for the denotation of a command. Sequencing composes state transformers. The local clause is

$$\mathcal{D}(\mathrm{local}(e, c))(s) = \mathrm{tail}(\mathcal{D}(c)(E(e, s) :: s)).$$

The operational account has a judgment $\langle c, s\rangle \Downarrow t$ and one rule for each command, with separate zero and nonzero branches. A local derivation evaluates its body in the extended store and returns the tail of the body's final store. These clauses and the corresponding functions are executable SODA definitions in `state-correspondence.soda`.

**Theorem 1** (correspondence)**.** For every command $c$ and stores $s, t$ in this core,

$$\langle c, s\rangle \Downarrow t \quad \Longleftrightarrow \quad \mathcal{D}(c)(s) = t.$$

*Proof.* The Lean development proves the forward direction by induction on operational derivations. It proves the reverse direction by induction on commands, constructing a derivation for the denotation. Sequencing uses the intermediate store given by the first command. The local case applies the induction hypothesis to the extended store and then discards exactly its first slot. Determinism follows from equality with the denotation. □

Two further theorems describe writes inside a local scope. An assignment to slot zero inside a local scope leaves the outer store unchanged on exit. An assignment to slot $i + 1$ updates outer slot $i$, using the expression's value in the extended environment. Restoring the whole incoming store on exit would violate the second property by discarding the outer update.

### 3.2 Axiomatic correspondence

Assertions are arbitrary predicates on stores. The Hoare system is defined and proved in Lean, with rules for skip, assignment, sequencing, zero-test branching, local scope and consequence. Its local rule requires the precondition to establish a body precondition in $E(e, s) :: s$; the body's postcondition applies the final assertion to the tail of its output store.

**Theorem 2** (Hoare soundness and relative completeness)**.** A triple $\{P\}\, c\, \{Q\}$ is derivable in this system exactly when every operational execution from a store satisfying $P$ finishes in a store satisfying $Q$. Equivalently,

$$\vdash \{P\}\, c\, \{Q\} \quad \Longleftrightarrow \quad \forall s.\ P(s) \Rightarrow Q(\mathcal{D}(c)(s)).$$

*Proof.* Soundness follows by induction on Hoare derivations and Theorem 1. For completeness, define the weakest precondition by $\mathrm{wp}(c, Q)(s) = Q(\mathcal{D}(c)(s))$. Induction on $c$ constructs the triple with this precondition; consequence then gives every valid triple. Relative completeness assumes arbitrary semantic assertions and proofs of the implication premises used by consequence. □

### 3.3 Executable weakest preconditions

To compute a precondition as a formula, we use an assertion language with arithmetic equality, truth, conjunction, disjunction and negation. The transformer $W(c, Q)$ follows Dijkstra's predicate-transformer approach [11]. Assignments substitute their expression for the assigned variable, sequencing calculates backwards, and branching combines the two preconditions under the corresponding zero or nonzero test. These operations remain within the assertion language.

Local scopes require two changes of environment. First, lift the postcondition's free indices by one so it refers to outer slots in the body's final store. Calculate the body's precondition under this lifted postcondition. Finally, substitute the initialiser for slot zero and decrement every outer index to return to the incoming environment. Writing $\uparrow Q$ for lifting and $\sigma_e$ for $0 \mapsto e$ and $i + 1 \mapsto \mathrm{var}(i)$ gives

$$W(\mathrm{local}(e, c), Q) = \sigma_e(W(c, \uparrow Q)).$$

The initialiser's variables remain in the incoming environment. Each nested scope applies these operations at its own boundary.

**Theorem 3** (exact syntactic preconditions)**.** For every command $c$, formula $Q$ and store $s$,

$$\mathrm{holds}(W(c, Q), s) = \mathrm{holds}(Q, \mathcal{D}(c)(s)).$$

Consequently $\{W(c, Q)\}\, c\, \{Q\}$ has a derivation in the Hoare system. Any valid precondition for $Q$ implies $W(c, Q)$.

*Proof.* The Lean proof uses a substitution lemma for arithmetic expressions and formulas, then induction on commands. Its scope case connects postcondition lifting to removal of the local slot, and entry substitution to the pushed initialiser. □

The transformer calculates syntax without executing $c$ on a store.

For example, initialise a local from outer slot zero and assign the local plus one to that outer slot. For the postcondition $\mathrm{var}(0) = 3$, the transformer calculates

$$W\big(\mathrm{local}(\mathrm{var}(0), 1 := \mathrm{var}(0) + 1),\ \mathrm{var}(0) = 3\big) = (\mathrm{var}(0) + 1 = 3).$$

The final formula refers only to the initial outer store. A write solely to the temporary local disappears from the precondition for an outer-store assertion. SODA executes this construction using ordinary semantic functions over the same command and expression constructors as its evaluators.

### 3.4 Connection to the executable definitions

We compare SODA's definitions with the Lean model by running 246 generated commands on six stores, giving 1,476 cases. The commands include every ordered pair of eleven atomic commands, branching and scope combinations, and nested locals. For every case, both SODA evaluators return the store calculated by the Lean denotation. Relational evaluation returns the same answer with a derivation that passes SODA's derivation checker.

Five postconditions per command give 1,230 symbolic preconditions. Both evaluators produce the formulas calculated by Lean. Evaluating these formulas in the six stores gives 7,380 comparisons; each agrees with the postcondition evaluated after execution.

We also introduced six faults into the semantic definitions. Three change execution by reversing sequencing, discarding outer updates, or evaluating an initialiser in the extended environment. The other three change precondition construction by reversing composition, omitting postcondition lifting, or failing to shift outer indices on scope entry. Each fault produces a disagreement on a concrete example. The derivation checker also rejects a scope derivation whose body premise has been removed.

### 3.5 Verified account transfers

The account-transfer program moves an amount between two balances. Transfers are sequential, and balances and amounts are natural numbers. A state contains a source balance $s$, destination balance $d$, requested amount $a$ and success flag. The program uses three nested locals to snapshot the balances and amount, then checks whether the transfer is affordable. The test $a \dot{-} s = 0$ is equivalent to $a \leq s$. On success it writes $s \dot{-} a$ and $d + a$ to the outer balances and sets the flag to one. On refusal it leaves both balances unchanged and clears the flag. The temporary snapshots disappear on scope exit.

The SODA function `transfer_program` constructs this command in the syntax shared by the operational judgment, denotation and weakest-precondition functions. A second function constructs a finite batch by setting the amount and invoking this transfer for each request. A later refused request leaves earlier successful transfers committed.

**Theorem 4** (ledger conservation)**.** For all natural balances, amounts and initial flags, the formal transfer has exactly the result

$$(s, d, a, f) \longmapsto \begin{cases} (s - a, d + a, a, 1), & a \leq s, \\ (s, d, a, 0), & a > s. \end{cases}$$

The success flag is one exactly when the request is affordable. The sum of balances is preserved by every transfer and by every finite list of requests. These properties hold for operational execution by the correspondence theorem. Applying the symbolic transformer to the conservation postcondition gives a precondition satisfied by every ledger input with the specified initial total.

*Proof.* Lean proves the single-transfer specification by reducing the command's denotation and applying natural-number arithmetic. Induction on the request list establishes batch conservation. Both proofs use the core's definitions of scope and sequencing. □

For an initial total $t$, the derived condition for one transfer is

$$\begin{aligned}&(a \leq s \ \wedge \ (s \mathbin{\dot{-}} a) + (d + a) = t)\\ \vee\, &(a > s \ \wedge \ s + d = t).\end{aligned}$$

An incorrect variant credits a fixed unit instead of the requested amount. The same transformer then produces the first branch with $d + 1$. At $s = 5$, $d = 4$, $a = 2$, the correct transfer ends at $(3, 6)$ and the incorrect one at $(3, 5)$. Its calculated precondition for total nine evaluates to false. The counterexample is checked by Lean reduction and reproduced by SODA.

We evaluated each transfer variant on 168 inputs and the batch program on 294 pairs of requests. In all 630 cases, both SODA evaluators agree with Lean on the command, final store, precondition formula and its truth value. The operational derivations pass the checker, and a direct Rust implementation gives the same final balances. All 462 cases using the correct transfer preserve the total; the faulty credit violates conservation on 64 inputs. We use innermost rewriting to avoid repeated evaluation of stored expressions.

### 3.6 Checking exported results

SODA exports a command, initial store and postcondition together with its calculated final store and precondition. A Lean program validates the answer by recomputing the denotation and precondition. The caller supplies the intended command, input and assertion separately, and the checker first compares them with those recorded in the result. Decoding checks constructors, arities and sorts against the core language, including its structural natural arithmetic.

**Theorem 5** (accepted results)**.** If the typed checker accepts evidence for expected syntax $(c, s, Q)$, its decoded final store $t$ and precondition $P$ satisfy

$$\langle c, s\rangle \Downarrow t \qquad \text{and} \qquad \forall u.\ \mathrm{holds}(P, u) = \mathrm{holds}(Q, \mathcal{D}(c)(u)).$$

The Hoare triple $\{P\}\, c\, \{Q\}$ is derivable. Acceptance also establishes that the evidence names exactly the caller's expected syntax. Conversely, every command, store and postcondition in the core has accepted canonical evidence.

*Proof.* Theorem 1 justifies the final store and Theorem 3 justifies the calculated precondition. Lean also proves encoder–decoder roundtrips for natural values, expressions, commands, stores and formulas. □

The result applies to core programs and stores of arbitrary finite size. The JSON reader limits nesting depth to 512 and rejects inputs that exceed it.

Across the 1,476 core cases and 630 ledger cases, both evaluators produce stores and preconditions accepted by the Lean checker, giving 4,212 accepted results. A nested-scope example supplied through the command-line interface gives two further accepted results. The checker rejects all 54 altered exports, which change programs, inputs, postconditions or constructor syntax. It also rejects all twelve results produced by the six faulty semantic definitions across the two evaluators.

The checker recomputes results independently of SODA's evaluators and precondition construction. Running it relies on the Rust parser and export adapter, JSON conversion, and Lean's compiler and runtime. The proofs use indexed store slots. SODA's named-binder substitution and the translation of arbitrary semantic modules remain unverified.

# 4 Implementation and semantic facilities

The shared representation supports sorted constructors, metavariables and lexical binders. Constructor, function and judgment signatures constrain the sorts of their arguments and results. Substitution avoids capture, and matching beneath binders supports alpha renaming. Users define executable rewrites, relational judgments and semantic functions over these terms.

Rewriting selects outermost or innermost contractions; compositional evaluation interprets arguments before applying a rule. Both evaluators handle conditionals by evaluating only the selected branch. Relational execution uses backward chaining, unification with an occurs check, and backtracking over alternative clauses. A separate checker validates each returned derivation by checking its rule instance, substitution, child derivations and side conditions. Search reports whether it exhausted the alternatives or reached a resource bound.

Equational certificates name the declared equations used at each step, their direction, the subterm position and the substitution. The checker matches the selected subterm against the equation's instantiated left side, up to alpha equivalence, and checks the replacement and final endpoint. Structural induction requires one case per datatype constructor and supplies induction hypotheses for immediate recursive fields. Checked certificates can be used as lemmas in later proofs.

Explicit finite ordered domains support semantic maps checked for totality, membership and monotonicity. Iterating a monotone map from bottom computes its least fixed point; finiteness ensures that the ascending sequence stabilises. Typed binders also let semantic functions return functions, including state transformers of the form $[\![e]\!] : S \to (A \times S)$. The bundled state library composes these functions by passing each updated store to the next computation.

SODA is implemented in Rust. Parsing and sort checking produce a module shared by evaluation, inference, proof checking, domain construction and state exploration. Evaluation and successor enumeration index rules by root symbol and arity while retaining declaration order. Inference uses corresponding clause indexes and caches rule-variable information. Evaluation and proof checking use the same capture-avoiding substitution implementation.

State exploration uses breadth-first search over contextual rewrites to find alternative transitions, cycles and shortest witness paths. Reachability and invariant checks operate on this graph; complete finite exploration supports safety checks, while a discovered violating path provides a counterexample. Syntactic analysis computes critical pairs for unconditional first-order rules and checks a decreasing-size termination criterion.

The command-line tools, editor support and web playground use the same semantic engines. Libraries supply the mathematical structures used in the examples, including natural numbers, rational arithmetic and computational effects. Modules can also be constructed through the AST library and embedded in Rust programs.

# 5 Case studies

Table 3 lists the additional semantic models written in SODA. We first examine compiler transformations involving overflow and undefined values, then summarise the other models and their checks.

| Model | Representation | Semantic operations |
|---|---|---|
| Typed lambda calculus | Typing and evaluation judgments | Lexical closures and application |
| Evaluation contexts | Decomposition and plugging relations | Ordered reduction |
| Finite CCS | Labelled transitions | Interleaving and synchronisation |
| Finite loops | Strict state-transformer tables | Least fixed points |
| Nondeterministic choice | Convex sets and Egli–Milner order | Choice and extension |
| Exceptions and state | Computation values and functions | Compositional effects |
| Probability | Rational subprobability weights | Choice and missing mass |
| Integer definedness | Finite bit-pattern sets and execution outcomes | Stable choices and directional refinement |

Table 3: Semantic presentations implemented in SODA.

### 5.1 Integer definedness and refinement

Compiler transformations must account for values whose behaviour is only partly specified. The integer model adapts the finite-set presentation in Alive2 [10]. It defines straight-line register programs with addition, unsigned division, selection and `freeze`, using SODA functions and the shared collection library. Widths range from one to eight bits. An undefined value denotes a set of possible bit patterns; poison and immediate undefined behaviour have separate constructors.

Repeated uses of an undefined register need not select the same value. With two-bit arithmetic, adding two uses of an undefined register can produce any of the four bit patterns. Freezing that register first selects one value for the execution. Adding that chosen value to itself can produce only zero or two. SODA's model represents the former by a set-valued register and the latter by separate executions with singleton values. A selection can ignore poison in an unselected arm, while undefined behaviour from an earlier instruction remains part of the execution.

Refinement requires every target execution in the supplied finite outcomes to be covered by a source execution. For an overflowing addition, removing the unsigned-no-overflow flag permits a concrete wrapped result and passes this comparison. Adding the flag introduces poison and fails it.

For widths one to three, we compare every operand pair in four addition modes and unsigned division with direct Rust calculations. Further examples cover partial undefined sets, stable and independent freezes, poison masking, immediate undefined behaviour and both directions of refinement. SODA's two evaluators agree on these demonstrations. These comparisons use supplied outcomes of finite register programs without memory. Extending them to all inputs and proving correspondence with Alive2 would require further work.

### 5.2 Other semantic models

The lambda-calculus model defines typing and call-by-value evaluation over shared syntax, using de Bruijn indices and explicit environments. Evaluating $(\lambda x.\lambda y.x)$ *true false* returns true because the inner closure retains the binding of $x$. A separate Boolean/pair language uses evaluation contexts to specify reduction order. The finite CCS model enumerates interleavings and synchronisation with a derivation for each successor; changing a channel removes a synchronisation, while restriction removes visible actions on that channel.

The finite loop model has bottom and three concrete stores. Its $4^3 = 64$ strict body maps each determine a loop denotation by least fixed-point calculation. For all 64 maps, the result agrees with an independent calculation that follows the body graph to termination, bottom or a repeated store. Comparing each result with every candidate fixed point also checks leastness.

The examples distinguish a decrementing loop that reaches zero from a repeated skip that diverges at nonzero stores.

The choice model checks the six nonempty convex subsets of a three-point chain against the quantified Egli–Milner order. The exception and state models check unit and associativity on representative computations and continuations. The probability model uses rational subprobability weights: a fair choice between true and divergence has true mass and total mass one half. Independent fraction calculations check the results of composing these computations.

# 6 Related work

**Executable semantics.** Redex supports executable semantic models with reduction exploration and randomised testing. Klein et al. [1] use these tools to examine models from programming-language research. SODA likewise executes semantic definitions and reports individual derivations. Its proof facilities check explicit equational steps and induction cases, while its finite-domain facilities construct ordered carriers and enumerate their maps.

K organises executable semantics around configurations, computations and rules [3]. Its reachability logic supports language-independent verification from operational rules [12]. SODA represents relational execution, state transformers and symbolic preconditions over a common object syntax. Lean proves correspondence for the stateful core, and the experiments compare SODA's results with that model.

**Metalanguages and proof support.** Ott compiles syntax, binding specifications, and judgments into proof-assistant definitions and typeset mathematics [2]. SODA executes rewrite rules and relational clauses directly, and checks equational certificates and structural inductions within the toolkit.

**Semantic foundations.** The operational examples follow the structural approach of Plotkin [4]. The denotational models implement instances of state transformers and fixed points [5], powerdomains [6, 9], and monadic accounts of computation [7, 8]. Dijkstra's weakest-precondition calculus [11] supplies the assertion-transformer construction. For local scopes in the stateful core, lifting and substitution make the symbolic precondition agree with the state transformer.

Alive2 uses SMT solving for bounded translation validation of LLVM transformations [10]. The SODA study adapts its integer definedness model as executable definitions, using explicit finite sets and outcome comparisons.

Lin et al. [13] generate matching-logic proof certificates for verification runs in K and check them using Metamath. SODA's result checker uses recomputation of the stateful core's denotation and precondition to apply the correspondence theorems to each accepted result.

# 7 Future work

Backward chaining can revisit the same goal in recursive judgments; tabling could retain answers and avoid repeated search. A verified translation of semantic modules would connect user-defined rules to the formal model used by the result checker. Extending the correspondence to loops requires a treatment of divergence and a choice of total or partial correctness.

# 8 Conclusion

SODA brings operational, denotational and axiomatic semantics into one executable language. Shared syntax and binding let language designers run definitions, inspect derivations and check proofs over the same programs. For the stateful core, Lean proves agreement between execution, state transformers and Hoare reasoning, including exact symbolic weakest preconditions. The account-transfer program uses these results to prove conservation of balances across arbitrary finite batches. A Lean checker validates the final stores and preconditions exported by SODA.

# References


[1] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. *Run Your Research: On the Effectiveness of Lightweight Mechanization*. POPL, 2012. Paper.

[2] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. *Ott: Effective Tool Support for the Working Semanticist*. ICFP, 2007. Paper.

[3] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

[4] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. 1981; reprinted in 2004. Paper.

[5] Dana Scott and Christopher Strachey. *Toward a Mathematical Semantics for Computer Languages*. Technical Monograph PRG-6, Oxford University Computing Laboratory, 1971. Publication record.

[6] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976. Paper.

[7] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Paper.

[8] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, LNCS 925, 1995. Paper.

[9] Claire Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science*, 1989. Paper.

[10] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of PLDI*, 2021. doi:10.1145/3453483.3454030.

[11] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. EWD 472.

[12] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. *Logical Methods in Computer Science*, 15(2), 2019. Article.

[13] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):56–84, 2023. doi:10.1145/3586029.