RESEARCH SURVEY AND FORMAL SEMANTICS

Recursive Improvement in Schmidhuber's Artificial Intelligence

Proof-carrying rewrites, effective semantics,
and a Lean specification of Gödel machines

Anonymous manuscript for *Higher Categorical Semantics*

Abstract

Jürgen Schmidhuber developed several mechanisms through which an artificial agent can change its future learning or search. This survey traces a line from self-referential learning and fast weights through the Optimal Ordered Problem Solver, Gödel machines and PowerPlay. Its formal focus is the Gödel machine, which commits to a timed rewrite after obtaining a proof that doing so has higher expected utility than continuing its current proof search. We model encoded programs by partial evaluation in the effective topos and represent certificates in slices over rewrite requests. A sound certificate projection factors through the subobject of true comparisons. Proof-labeled rewrites compose as spans with elapsed-time labels. A run-wide guarantee follows from preservation of live checker soundness, and an explicit countermodel shows that the first useful rewrite can destroy that property. A Lean 4 specification verifies the checkpoint rule, the invariant argument and the countermodel. A separate reduction from the halting problem rules out a total exact improvement test for arbitrary programs in a bounded-reward, infinite-horizon setting.

Keywords. Recursive self-improvement, Gödel machines, effective topos, proof certificates, bicategories of spans, Lean.

1 Levels of self-improvement

A learner can improve a prediction, change how it learns the next prediction, or replace the process by which it decides to change its learning rule. These are distinct operations even when a single machine implements them all. Let x be task state, m a representation of the current learner and k a representation of the rule for updating m. An update $(x, m, k) \mapsto (x', m', k)$ keeps the rule fixed. An update with k' in the last component changes the rule available at the next step. A system whose writable image also contains a proof searcher and its checking code can make further changes within the same address space. The useful question is what the live system must establish before it adopts a change.

Schmidhuber's work supplies different answers. Online meta-learning can make successful changes persist under a reward test. Program search can freeze solutions and exploit them as primitives or prefixes on later tasks. A proof-driven agent can compare a proposed installation with continued search and change the very searcher that made the comparison. An open-ended learner can add a task only after its proposed solver has demonstrated retention of earlier tasks. Neural fast weights provide a further mechanism in which learned parameters determine how a writable weight state updates. The Gödel machine makes the issue of recursive authorization especially precise. Its current search and proof machinery can appear in both the proposed change and the forecast of what happens if the change is postponed. [1, 2, 3, 4, 5, 6]

The historical claims below follow the cited sources. The semantics, its comparison counterexamples and the Lean specification are constructions of this manuscript. The formal treatment separates three mathematical levels. An interpreter assigns partial behavior to writable code. A model assigns values to counterfactual branches from the same snapshot. A verification relation records why a particular branch is authorized. A quotient at one level may erase information that the next level uses. For example, two programs that return the same answers can incur different runtimes and can occupy different roles in a proof about what the machine will do next.

2 Research line from meta-learning to proof-driven rewrites

2.1 Self-referential learning, history compression and fast weights

The 1987 diploma thesis frames meta-learning as the joint problem of solving tasks and improving the strategies by which tasks are solved. It proposes feeding learning strategies back into the learning process, so that higher levels can themselves be learned. The constructive issue is representation. A change that the learner can express as data can be generated, tested and retained by the same system. The thesis explores symbolic and associative forms of this idea. [1]

In the neural work, a slow network can learn to control rapid changes in a fast-weight memory. A subsequent self-referential weight-matrix design makes the writable weights participate in computing their own updates. This line also connects to Schmidhuber's work on history compression, where a lower recurrent predictor passes unexpected events to a slower level. History compression concerns temporal representation and the allocation of learning across timescales. It supplies a different use of hierarchy from rewriting an updater. Hochreiter and Schmidhuber's LSTM addresses learning long-range dependencies through gated recurrent memory. LSTM's memory updates and gradient training, in their original form, operate within a given training rule. Schmidhuber's later survey places these results within the broader history of deep and recurrent learning. [7, 8, 9, 10, 11]

Schlag, Irie and Schmidhuber relate linearized self-attention to a fast-weight programmer. Irie, Schlag, Csordás and Schmidhuber develop a modern self-referential matrix whose current weights produce quantities used in their next update. The general state-space form is

$$y_t = g(W_t, x_t), \qquad W_{t+1} = F(W_t, x_t).$$

Here the dependence of F on the writable W_t carries the self-reference. The surrounding tensor operations specify F, and an outer training process can shape the initial weights. This is a learned update architecture with experimental evidence in the cited work. Its mathematical question is how useful update behavior arises and transfers across tasks. A Gödel machine asks whether a specific change at a specific time dominates a particular continuation under a stated utility model. [12, 6]

2.2 Learning models of the environment

In *Making the World Differentiable*, Schmidhuber uses a learned recurrent model of environmental dynamics to provide credit-assignment paths for a controller. The 1991 curiosity proposal adds a reason to choose actions that improve the model's knowledge. The architecture can therefore direct experience toward useful changes in its predictor. In a later empirical study, Ha and Schmidhuber fit a generative recurrent model of the environment in a compressed latent representation, evolved a controller inside that model, and transferred the controller back to the actual environment. These

studies concern learning and using a predictive model. The Gödel-machine comparison additionally asks which facts about such a model justify replacing the search or controller that uses it. [13, 14, 15]

Schmidhuber's Speed Prior concerns the cost of generating possible observations. It favors data with fast computational descriptions and supports computable prediction under its stated assumptions. This role for resources complements the separate use of a computational budget in program search. In prediction, the bias ranks explanations of data. In proof search, a bias ranks programs that attempt to build evidence. The external utility of the agent can in turn charge for the time spent obtaining that evidence. [16, 3, 4]

2.3 Reward tests, retained programs and invented tasks

The Success-Story Algorithm of Schmidhuber, Zhao and Wiering connects persistent changes to observed long-term reward. Its acceptance condition uses the agent's measured reward history and a stack of modification checkpoints. When the condition fails, the system can undo changes. The method supplies an operational response to credit assignment across changes of inductive bias. A reward test of an observed history and a proof about two future trajectories have different inputs and different inferential strength. [2]

The Optimal Ordered Problem Solver (OOPS) applies bias-optimal program search to a sequence of tasks. Successful code is frozen and available to later programs. A later search can build on the most recently frozen solution or start with a fresh candidate that can reuse earlier code. The schedule protects some search time for both routes. The benchmark of bias-optimality relates a candidate's test time, its search probability and the available budget. For a candidate of weight $P(p \mid r) > 0$ and test cost $c(p, r)$, the relevant scale is $c(p, r)/P(p \mid r)$, up to the overhead of the specified search. The guarantee is indexed by the representation, bias and cost model. In particular, freezing code preserves an available computational resource, and a prefix can affect how suffixes are explored. [3, 17]

PowerPlay searches for a pair consisting of a task and a solver modification. In its strict retention variant, the candidate must solve the new task and all retained tasks, while the incumbent fails the proposed novelty criterion. A task can also require achieving a familiar result under a tighter resource bound. The search order biases the system toward pairs it can find and validate quickly, given its accumulated experience. Schmidhuber connects this process to intrinsic motivation. In his compression-progress account, an agent can receive intrinsic reward for making its representation of accumulated data better, which gives it an incentive to acquire data it can learn to compress. The first experiments with PowerPlay give evidence for specific constructions and task regimes. [5, 18, 19]

Table 1: The object changed and the evidence used to keep the change.

Research strand	Writable or extensible component	Basis for accepting or retaining the change
Success-Story Algorithm	Inductive bias and stacked modifications	Observed reward history with rollback
OOPS	Repository of frozen programs and search bias	Task verification and a budget indexed by candidate bias
Gödel machines	Executable image, including proof search and formal theory	A checked target theorem comparing a timed rewrite with continued search
PowerPlay	Solver and audited task repertoire	Validation of a new task together with retention of recorded skills
Self-referential weights	Fast weight matrix that influences its next update	A learned update rule shaped by the outer training objective

2.4 The Gödel machine

Schmidhuber's Gödel-machine manuscript first circulated in 2003. We use the December 2006 revision, especially its description of the initial program and Theorem 4.1. Its machine interleaves ordinary problem solving with a search over programs that construct proofs. The initial formal system describes the machine, initial software, utility and available knowledge about the environment. A proof technique can propose a *switch program* and produce a derivation of a target theorem about the two settings of a switch bit at a specified future time. The check instruction tests the form of the theorem and whether it can set the bit in time. At the scheduled point, control passes to the switch program. The program can modify the solver, the proof searcher or the formal system used in subsequent search. The design thus treats code that determines future acceptance decisions as writable. The interpreter and hardware-reserved registers define the boundary of the writable image. [4]

The initial proof search can use a bias-optimal scheduling idea, adapted to proof techniques that read a changing machine state. Universal search concerns the time to discover a proof technique with suitable bias and running time. The target theorem concerns remaining utility, including the cost of that discovery. These criteria can favor different choices. Hutter's universal artificial intelligence supplies related work on sequential decision under algorithmic priors. The Gödel-machine paper discusses a resource-bounded variant of Hutter's agent as a possible initial solver. The self-rewrite mechanism then evaluates changes relative to the selected initial software and objective. [17, 4, 20]

3 Operational model and the meaning of the comparison

3.1 Snapshots, schedulability and continuation value

Let S be a type of checkpoint snapshots and let $P = \mathbb{N}$ be a space of program codes. A snapshot s records a clock $t(s)$, the code of the policy, searcher, theory and checker, and the working memory and observation history required by the model. A rewrite $r = (p, \tau)$ pairs executable code with a scheduled checkpoint. Write $B \hookrightarrow S \times P \times \mathbb{N}$ for admissible requests. Its conditions include $t(s) < \tau$, enough time to complete the gate, and the ability to bind the proof to the actual program and snapshot. A temporal implementation can check a verified projection of the snapshot or a formally

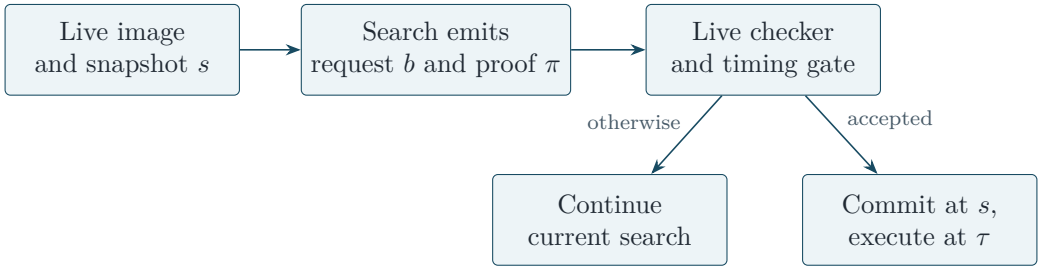


Figure 1: One decision checkpoint. The switch and wait branches in the value comparison include their respective future trajectories.

specified condition on the state at τ. The Lean model uses exact snapshot equality at the gate and a separate schedulability check.

For a request $b = (s, r) \in B$, the branch $\mathsf{commit}(r)$ commits at s and schedules r. The branch wait discards the offered switch and continues under the actual search regime, including any later authorized switches. Both branches start with the same information and use the same external objective and environment law. The source paper compares the switch-bit settings at t_1 and transfers control after its timed check. Our checkpoint notation groups the gate, timing conditions and transfer into a single macro transition. A model of this macro transition must account for intervening costs and observations.

For finite horizon H, a typical value functional for branch a is

$$V(s, a) = \mathbb{E}_s^a \left[\sum_{j=0}^{H-1} \gamma^j (R_j - C_j) \right], \qquad 0 < \gamma \leq 1. \tag{1}$$

The conditional law includes the hidden environment state through the belief at s, when the environment is partially observed. R_j is external reward and C_j denotes any computation or action cost used by the objective. If the model already represents resource use through latency and foregone rewards, an additional charge requires justification. For an infinite horizon with $\gamma < 1$ and bounded net rewards, the same expression is absolutely convergent. The semantic judgment on B is

$$J(s, r) \quad \Longleftrightarrow \quad V(s, \mathsf{commit}(r)) > V(s, \mathsf{wait}). \tag{2}$$

Values can be rational for a finite rational model. With a general environment the values can be defined externally in **Set**. The later topos construction uses an internal truth subobject only when the value relation is given internally.

3.2 The guarantee and two stronger claims

Theorem 4.1 of the source paper calls a rewrite globally optimal in the specified sense of beating continued proof search. It states a consistency assumption for the formal system. For a conclusion about the chosen external model, we state the exact semantic premise. A certificate accepted for b must entail $J(b)$ in that model. A syntactically consistent theory has a model, while truth in the intended model requires that its relevant assertions and inferences hold there. In particular, a proof using an environmental axiom transfers to the actual environment only when the axiom is valid for that interpretation. [4]

Let E_i partition the wait trajectories by events determined at bounded stopping checkpoints σ_i. Suppose each event has probability $p_i > 0$, and for simplicity the stopping offset is the constant n_i on E_i. On a finite horizon, a partition can include an event on which search never switches by the deadline. Define a_i to be the conditional expected discounted net reward before σ_i. Define w_i to be the conditional expected reward from σ_i onwards, using discount origin σ_i. The tower rule gives

$$V(s, \mathsf{wait}) = \sum_i p_i(a_i + \gamma^{n_i} w_i). \tag{3}$$

Equation (2) can therefore dominate a policy that has access to many future alternatives even if a realized trajectory under that policy has a higher payoff. For example, take wait payoffs 12 with probability 1/4 and 4 with probability 3/4, after accounting for search cost. Their expectation is 6. A certain commit payoff of 8 passes the target test, while the favorable wait outcome pays 12.

The admissible requests at s may also contain a different switch program with value 10 that the current search will never propose in the remaining time. The comparison of 8 with 6 leaves this other request unranked. Maximizing over $\{\mathsf{commit}(r), \mathsf{wait}\}$ and maximizing over all computable replacement policies require different choice sets. The checkpoint rule establishes the former. In addition, values at different checkpoints have different residual horizons, observations and opportunity sets. To telescope local inequalities into one objective over a sequence requires a common potential or an explicit policy-improvement argument.

4 Programs and truth in the effective topos

4.1 Partial evaluation and intensional self-reference

Work in the effective topos **Eff**, with natural numbers object $\mathbb{N}$. Encode finite machine descriptions, histories and proof texts by natural numbers. For the following construction, state and output types X, Y have fixed effective encodings. Let $\mathsf{Run}(p, x, n, y)$ be a primitive recursive relation saying that the deterministic interpreter has returned y from code p and input x within n steps. Then

$$D = \{(p, x) \mid \exists n, y\, \mathsf{Run}(p, x, n, y)\} \hookrightarrow \mathbb{N} \times X$$

is the domain of the universal partial evaluator. Determinism gives a unique y for each point of D. The functional relation defines the right leg of a partial map

$$\mathsf{eval} : \mathbb{N} \times X \longleftarrow D \longrightarrow Y.$$

Partial maps compose by pulling back the domain of the second computation along the result of the first. With a step bound, the interpreter can instead return a result or the suspended machine state at the end of the budget. That is the interface needed for dovetailing proof techniques. Hyland's realizability construction supplies the base topos. Moggi's computational semantics supplies related background on separating values from computations. [21, 22]

For a total computable transformation $f : \mathbb{N} \to \mathbb{N}$ of program codes, Kleene's recursion theorem supplies a code q with $\varphi_q \simeq \varphi_{f(q)}$, where $\simeq$ is equality of partial behavior. In the usual acceptable numbering, let $s(a, b)$ be a specialization function satisfying $\varphi_{s(a,b)}(x) \simeq \varphi_a(b, x)$. Choose e to index the partial computation $(b, x) \mapsto \varphi_{f(s(b,b))}(x)$. Then $q = s(e, e)$ satisfies the equation. This explains how an executable description can uniformly access a usable index for itself. Proofs about run time, source locations and writable registers require the intensional code and interpreter in addition to

the partial-behavior equation. Kavvos develops categorical models of intensional recursion and gives a related account of this distinction. [23, 24]

The recursion theorem provides self-reference. Authorization is a separate relation on descriptions and states. For example, let two codes produce the same answers and let one wait one extra step before every return. They define the same uninstrumented partial function. With a reward for returning before a deadline, the codes can yield different values in Equation (1). A quotient of codes by uninstrumented behavior would identify evidence needed by the proof checker. One can carry extra observations in the semantic codomain when a specified equivalence is useful.

4.2 The family of certificates over admissible requests

Write c_s for the live checker and T_s for the live object theory. Define the object of syntactically accepted certificates by

$$C = \{(b, \pi) \in B \times \mathbb{N} \mid \mathsf{Check}(c_s, T_s, b, \pi)\}, \qquad q : C \longrightarrow B. \tag{4}$$

Here $b = (s, r)$ and π is a derivation code. The check verifies that π concludes the instantiated target and is accepted in the current theory. A resource-bounded check gives an effective predicate. One may instead use a partial checker and include the bounded computation trace of a successful check among the witnesses. The object q in the slice $\mathbf{Eff}/B$ distinguishes accepted derivations of the same claim. The image $\mathrm{im}(q) \hookrightarrow B$ forgets which derivation established acceptance.

When truth $J \hookrightarrow B$ is specified internally, the comparison has a simple characterization in the slice. For example, a finite rational transition model with internally represented finite histories and rational values defines J by rational strict order. The same statement works externally in **Set** for a real-valued environment model and its corresponding syntactic certificate family.

Proposition 4.1 (Soundness and change of context)**.** *Let $j : J \hookrightarrow B$ be the subobject of true comparisons and $q : C \to B$ the projection of accepted certificates. Semantic soundness is equivalent to the existence of a unique map $a : C \to J$ such that $ja = q$. It is also equivalent to* $\mathrm{im}(q) \subseteq J$*. Both the factorization and the acceptance family reindex along any map $f : X \to B$.*

Proof. Soundness is the internal sequent $\pi \in C_b \vdash b \in J$, interpreted in the context C. Equivalently, q lands in the subobject J. The landing map is unique because j is monic. The image of q is the least subobject through which q factors. Pulling the diagram back along f gives $X \times_B C \to X \times_B J \to X$, and regular image factorizations in a topos are stable under pullback. □

This is a statement about certificate families under substitution. For each $X \to B$, one obtains a family of proof witnesses for the requests indexed by X. Different proofs may have distinct search or verification costs. Taking the image changes that family into the predicate that some proof is accepted. The reverse inclusion $J \subseteq \mathrm{im}(q)$ would assert semantic completeness for the selected comparison domain. Even if this inclusion held on a finite domain, finding a witness before its activation deadline would require a further scheduling argument. The slice fibration gives the uniform change-of-context rule used here. [25]

5 Composition of rewrites and preservation of soundness

5.1 Proof-labeled spans

Let R be the object of eligible commit events, with a point (s, r, π, s') carrying an accepted certificate and the resulting snapshot after the macro step. Its maps

$$S \xleftarrow{d_0} R \xrightarrow{d_1} S, \qquad \ell : R \to \mathbb{N}$$

record the source, target and elapsed time. A partial execution relation can be represented by including an execution witness with s'. For the deterministic macro interpreter, a given selected event has one target. All eligible events can form a span even when the actual search selects just one of them. The two uses support different questions. The former describes available proof-carrying rewrites. The latter determines which execution the machine follows. For traces that interleave search and installation, take the coproduct of R with a wait span recording ordinary checkpoint transitions. Its two endpoint maps and elapsed-time label are defined componentwise. The actual run then selects a compatible sequence of edges in this combined span.

In a category with finite limits, spans are 1-cells and maps commuting with their legs are 2-cells. Take the maps of spans here to preserve ℓ. The composite of two compatible spans has pullback apex

$$R_1 \times_S R_2 = \{(e_1, e_2) \mid d_1(e_1) = d_0(e_2)\}, \qquad \ell(e_1, e_2) = \ell_1(e_1) + \ell_2(e_2).$$

The equality in the pullback is equality of snapshots, which include the live checker and theory. Thus the second witness is checked under the image left by the first event. A commuting map of spans can, for example, normalize the format of proof witnesses while preserving endpoints and elapsed time. Standard span composition supplies the bicategorical structure. [26]

Proposition 5.1 (Coherent finite composition)**.** *For any finite composable sequence of proof-labeled spans, all parenthesizations have canonically isomorphic apices and equal elapsed time on corresponding witnesses. These comparison isomorphisms satisfy the associativity pentagon and unit triangles.*

Proof. A map into any iterated pullback is a compatible family of maps into the original span apices. Thus each parenthesization represents the same compatible-family functor. The unique comparison preserving the projections gives the associator. Any two paths around the pentagon preserve every projection, so uniqueness makes them equal. The identity span has diagonal endpoint maps and zero elapsed time. This gives the unit maps and the triangle by the same argument. Addition of natural numbers makes all the comparisons label preserving. □

The image of (d_0, d_1, ℓ) provides a relation on the endpoints and elapsed time. There is a 2-cell from the original span to that image span, given by the image projection. In general this 2-cell is noninvertible. In **Set**, a two-element witness set and a one-element witness set over the same endpoints and cost already give an example. The relation can answer whether a transition of a given duration is available. The span also supports counting or transporting its proof witnesses. Coherence of span composition gives a semantics for sequences of self-changes. A separate invariant is required to carry the utility guarantee through such a sequence.

5.2 Soundness of the live checker along a run

Let $F : S \to S$ be the selected deterministic macro step, including the continue-search case. An environment stream or seed can be included in the state to make such a step deterministic for a particular run. Define $\mathsf{Sound}(s)$ to mean that the checker and theory at s are sound for the target comparisons at s. Let $I \hookrightarrow S$ satisfy

$$I(s) \Longrightarrow \mathsf{Sound}(s), \qquad I(s) \Longrightarrow I(F(s)).$$

A startup state in I may evolve to a state with different proof software. The closure premise refers to both waiting and committing, and can be established by a proof about a particular rewrite policy.

Theorem 5.2 (Soundness along a certified run)**.** *For $s_0 \in I$, every commit r taken at $F^n(s_0)$ has strictly greater value there than continued search from the same checkpoint.*

Proof. Induct on n using closure to show $I(F^n(s_0))$. If the selected branch is a commit, the gate has accepted a certificate for the current snapshot and rewrite. Apply soundness at $F^n(s_0)$ and Equation (2). □

Proposition 5.3 (Greatest invariant inside an audit predicate)**.** *For an audit predicate $A \hookrightarrow S$, let*

$$A^\infty(s) \quad \Longleftrightarrow \quad \forall n \in \mathbb{N},\ A(F^n(s)).$$

This is the largest F-invariant subobject contained in A. The formula also defines it internally when F and A are internal and F^n is obtained by natural-number recursion.

Proof. The case $n = 0$ gives containment. Since $F^n(F(s)) = F^{n+1}(s)$, the predicate is closed under F. If $I \subseteq A$ is F-invariant, induction gives $I(s) \Rightarrow A(F^n(s))$ for each n, proving maximality. □

For a nondeterministic transition relation, a corresponding greatest invariant consists of states for which every finite reachable path stays in the audit predicate. With positive-probability branches, a requirement that all modeled branches remain sound can be expressed by requiring closure under each supported successor. These are properties of an operational model. Computing the greatest invariant, or certifying that a particular initial state belongs to it, can require solving a separate verification problem.

5.3 How the other architectures fit the semantics

OOPS. Order versioned code repositories by extension of the frozen segment. Each extension induces an embedding of code with resolved references in the old repository into code available in the new one. With stable address interpretation, repeated extension composes these embeddings. The functor of available code records how earlier solutions remain usable. Search probabilities and verification times live over each repository and can change with it. The certification layer for OOPS tests whether a program solves a task, whereas the Gödel comparison also evaluates the search process that would replace the candidate. [3, 4]

PowerPlay. For a solver p, let A_p be the tasks it passes under the chosen validator, including resource bounds encoded in the tasks. A validated state has a finite record $D \subseteq A_p$. In a strict retention step to p', a new task d witnesses $d \notin A_p$ and $D \cup \{d\} \subseteq A_{p'}$. Sending a state to its record D projects these transitions to a poset of finite subsets ordered by inclusion. Repeated strict

retention implies that every earlier record is contained in the final record and that the final solver passes it. This guarantee is about the recorded tasks and their validation criteria. Generalization outside that record is a separate empirical question. [5, 18]

Self-referential weights. For finite-precision weights, the input-output update of Section 2.1 can be written as a Mealy coalgebra $W \to (Y \times W)^X$ whenever the update is total, including whatever behavior the numerical semantics specifies on exceptional inputs. The state W contains data used to compute its successor. The coalgebra expresses continual adaptation under a learned update rule. A checker, target judgment and timed commit protocol would be additional structure on this neural system. [6]

6 Lean specification of the Gödel-machine checkpoint

6.1 Executable gate and specification boundary

The companion file `GodelMachine.lean` uses Lean 4.24.0 and the core distribution. A writable `Image` contains codes for the task policy, proof searcher, object theory and checker. `Runtime` supplies the checkpoint interpreter. Its `search` and `check` operations use the live code numbers. A `Proposal` carries a claim naming the snapshot and rewrite, together with a derivation code. The decision function returns `commit` exactly when the search offers a proposal and the proposal is accepted. Its four tests require equality with the live snapshot, a future activation time, schedulability, and acceptance by the live checker for the live theory. The step function either continues search or executes the selected rewrite, which can change all four codes. [27]

The executable gate operates on syntax, time and a verifier result. `Target` supplies the independent value judgment from Equation (2). Its value type is a parameter with a strict comparison, and the instance used in the example is the integers. `SoundAt` makes semantic soundness a hypothesis about the interpreter and value model at a given snapshot. It ranges over all claims for that snapshot, including those screened out by a different gate test. The following declarations connect the value judgment to the executable step.

```
def Target {U : Type} [LT U] (value : Snapshot → Choice → U)
    (s : Snapshot) (r : Rewrite) : Prop :=
  value s .keepSearching < value s (.commit r)

def SoundAt {U : Type} [LT U] (rt : Runtime)
    (value : Snapshot → Choice → U) (s : Snapshot) : Prop :=
  ∀ (r : Rewrite) (proofCode : Nat),
    rt.check s.image.checker s.image.theory ⟨s, r⟩ proofCode = true →
      Target value s r

def step (rt : Runtime) (s : Snapshot) : Snapshot :=
  match choose rt s with
  | .keepSearching => rt.continueSearch s
  | .commit r => rt.executeRewrite s r
```

The callbacks abstract the instruction-level interpreter, proof-technique scheduler and the external model of remaining utility. An implementation that uses the specification must relate its actual search, timed transfer and executions to these callbacks and establish `SoundAt` for the relevant claims. The schedulability check abstracts the original machine's time check, while the value model accounts for the consequences of scheduling and execution. The Lean theorem is a

Table 2: Principal results checked in the Lean companion.

Declaration	Statement
`commit_has_certificate`	A commit came from the live searcher and passed all four gate tests.
`choose_of_accepted`	An offered and accepted proposal selects its indicated rewrite.
`stale_offer_waits`, `expired_offer_waits`	The gate keeps searching on a mismatched snapshot or a missed checkpoint.
`accepted_commit_better`	Soundness of the live checker implies the strict comparison for a commit.
`every_commit_better`	A step-closed soundness invariant gives that comparison at every commit.
`core_greatest`	The greatest invariant contained in an audit predicate consists of states that pass the audit at every finite run checkpoint.
`Example.no_preserved_soundness`	A sound initial checker can authorize a beneficial rewrite that destroys future soundness.

checkpoint specification with conditional correctness. The bicategory and the unbounded-horizon undecidability proof are developed mathematically in this manuscript.

6.2 A good rewrite can break the next proof system

The example starts from a live checker with code 40, which only accepts a designated first candidate with code 1 and its matching theory and certificate. The specified branch values at that state are 8 for this candidate and 6 for continuing. The accepted rewrite moves the machine to checker 41, which accepts every claim. The successor search offers a second candidate. Its branch value is 2, while continued search at that checkpoint has value 5. The search thus takes a harmful second rewrite. Each comparison uses the same snapshot for both branches. A finite undiscounted model realizes these values at the two checkpoints. Let the first commit pay six before the successor, and let the successor's selected commit pay two. Waiting one tick initially incurs a cost of two, after which the first commit is still selected before expiry. Waiting at the successor pays five and lets the second offer expire. The two initial values are then $6 + 2 = 8$ and $-2 + 6 + 2 = 6$.

Checkpoint	Checker	Candidate	Commit	Wait	Outcome
Initial	40	1	8	6	Sound acceptance
After rewrite 1	41	2	2	5	False acceptance

The example verifies both gate decisions and checks the two value comparisons by reduction and proof. It also proves that the starting state lies outside the greatest invariant contained in live soundness, even though it starts with a sound checker. This separates the premise of the local comparison theorem from the invariant needed for Theorem 5.2. A run-wide guarantee requires a rule for transferring soundness to the successor. Such a rule can be represented by a closure hypothesis, for example an independently justified condition on permissible verifier replacements.

To reproduce the check with Lean 4.24.0, run

```
lean GodelMachine.lean
```

The file elaborates all declarations and evaluates both commit choices and the first installed image. The axiom reports printed by Lean audit the principal proofs. The two comparison theorems, maximality of the invariant core and failure of preserved soundness in the example all have empty axiom dependency reports.

7 Computability limits and trust transfer

Theorem 7.1 (No total exact test for the unbounded class)**.** *Fix rational $0 < \gamma < 1$. For a class containing arbitrary computable deterministic agents with infinite horizon and net rewards in $\{0, 1\}$, there is no total computable test that decides Equation (2) for all its instances. This holds even with a fixed zero-valued wait policy and a single candidate rewrite per instance.*

Proof. Given a program e and input x, effectively construct a candidate that, after activation, simulates $e(x)$ and emits reward 1 exactly once if that simulation halts. Every other reward is zero. A run of the fixed wait policy yields zero at every step. If $e(x)$ halts and the payoff occurs at finite time T relative to the checkpoint, the commit value is $\gamma^T > 0$. If it runs forever, the commit value is zero. Thus the exact strict comparison would decide the halting problem. [28] □

For this reduction, even the value is uniformly approximable. Simulate to a cutoff N. If no reward has appeared, any later payoff is at most γ^{N+1}, which tends effectively to zero. The obstruction comes from exact strict comparison at zero. A procedure may still certify many profitable changes, and the construction yields a semidecision of profitability for this particular family. For an explicit finite-horizon model with finite branching and rational probabilities and rewards, evaluation of a fixed candidate and wait policy can instead be done by finite enumeration. These results scope the impossibility claim to exact total decision over the unbounded class and permit sound partial certification.

Changing a proof system raises a second issue. Let T be a recursively axiomatized theory extending enough arithmetic for its usual provability predicate Pr_T to satisfy the derivability conditions. Löb's theorem states that if $T \vdash \mathrm{Pr}_T(\ulcorner\phi\urcorner) \to \phi$, then $T \vdash \phi$. A consistent T consequently cannot establish its own unrestricted reflection scheme, since the instance $\phi = \bot$ would yield $T \vdash \bot$. A rewrite can still have a proof about the soundness of a particular restricted checker, a finite class of target statements, or a weaker theory. Such a proof needs its own justification. The invariant formulation records exactly where that justification enters. [29]

8 Conclusion

Schmidhuber's research supplies distinct mechanisms for feeding learning and search back into their own operation. The Gödel machine gives this idea a specific control rule. It searches for a proof of a counterfactual comparison, binds that proof to a scheduled rewrite, and can change the code that makes future comparisons. The effective topos interprets partial computation on those codes. The slice over admissible requests tracks the evidence used to authorize a switch, and span composition tracks how the resulting machine becomes the source of the next event. The Lean specification isolates the conditional guarantee. A sound live checker certifies a better branch at its checkpoint, and preservation of live soundness makes that conclusion available at each subsequent commit.

Authorship, automation and evidence. The manuscript uses an anonymous review byline. ChatGPT assisted with literature synthesis, drafting and development of the Lean specification. The supporting evidence consists of the cited publications, the mathematical proofs in the manuscript and the Lean checks described in Section 6.

References

[1] J. Schmidhuber. *Evolutionary Principles in Self-Referential Learning*. Diploma thesis, Technische Universität München, 1987. Author's scan.

[2] J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130, 1997. https://doi.org/10.1023/A:1007383707642.

[3] J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004. arXiv:cs/0207097. https://doi.org/10.1023/B:MACH.0000015880.99707.b2.

[4] J. Schmidhuber. Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements. Technical report IDSIA-19-03, 2003, revised December 2006. arXiv:cs/0309048v5.

[5] J. Schmidhuber. PowerPlay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 4:313, 2013. https://doi.org/10.3389/fpsyg.2013.00313.

[6] K. Irie, I. Schlag, R. Csordás, and J. Schmidhuber. A modern self-referential weight matrix that learns to modify itself. *Proceedings of the 39th International Conference on Machine Learning*, PMLR 162:9660–9677, 2022. PMLR 162, Irie et al.

[7] J. Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992. https://doi.org/10.1162/NECO.1992.4.1.131.

[8] J. Schmidhuber. A 'self-referential' weight matrix. In *ICANN '93*, pp. 446–450. Springer, 1993. https://doi.org/10.1007/978-1-4471-2063-6_107.

[9] J. Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992. https://doi.org/10.1162/NECO.1992.4.2.234.

[10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735.

[11] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. https://doi.org/10.1016/j.neunet.2014.09.003.

[12] I. Schlag, K. Irie, and J. Schmidhuber. Linear transformers are secretly fast weight programmers. *Proceedings of the 38th International Conference on Machine Learning*, PMLR 139:9355–9366, 2021. PMLR 139, Schlag et al.

[13] J. Schmidhuber. *Making the World Differentiable: On Using Self-Supervised Fully Recurrent Neural Networks for Dynamic Reinforcement Learning and Planning in Non-Stationary Environments*. Technical report FKI-126-90, Technische Universität München, 1990. Author's scan.

[14] J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *From Animals to Animats*, pp. 222–227. MIT Press, 1991. Author's copy.

[15] D. Ha and J. Schmidhuber. Recurrent world models facilitate policy evolution. *Advances in Neural Information Processing Systems*, 31, 2018. NeurIPS proceedings.

[16] J. Schmidhuber. The Speed Prior: A new simplicity measure yielding near-optimal computable predictions. In *Proceedings of the 15th Annual Conference on Computational Learning Theory*, pp. 216–228. Springer, 2002. Author's copy.

[17] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973. Math-Net.Ru record.

[18] R. K. Srivastava, B. R. Steunebrink, and J. Schmidhuber. First experiments with PowerPlay. *Neural Networks*, 41:130–136, 2013. https://doi.org/10.1016/j.neunet.2013.01.022.

[19] J. Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010. https://doi.org/10.1109/TAMD.2010.2056368.

[20] M. Hutter. *Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability*. Springer, 2005. https://doi.org/10.1007/b138233.

[21] J. M. E. Hyland. The effective topos. In A. S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*, pp. 165–216. North-Holland, 1982. Author's scan.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. https://doi.org/10.1016/0890-5401(91)90052-4.

[23] S. C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3(4):150–155, 1938. https://doi.org/10.2307/2267778.

[24] G. A. Kavvos. On the semantics of intensionality and intensional recursion. 2017. arXiv:1712.09302.

[25] T. Streicher. Fibered categories à la Jean Bénabou. 2018. arXiv:1801.02927.

[26] M. Shulman. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories*, 20(18):650–738, 2008. TAC article.

[27] Lean FRO. Lean 4.24.0 release notes. 2025. Lean language reference.

[28] W. Dean and A. Naibo. Recursive functions. *Stanford Encyclopedia of Philosophy*. SEP entry.

[29] R. Verbrugge. Provability logic. *Stanford Encyclopedia of Philosophy*. SEP entry.