Executable Σ-Types: A Runtime for Proof-Carrying State Transitions via Gödel Encoding

Adrian Diamond

April 26, 2026

Abstract

Modern verification systems often separate static proof from runtime execution. Dependently typed languages ensure correctness before execution, while operational semantics define execution without necessarily carrying intrinsic proof objects. We propose a runtime architecture in which each execution step produces a dependent pair consisting of a next state and a proof of its validity. This yields a system in which correctness is not merely checked prior to execution, but constructed during execution. We further propose a Gödel-encoded representation of transition rules, allowing runtime transitions to be treated as machine-checkable syntactic objects.

Contents

1 Introduction

Modern verification systems separate static proof from runtime execution. Dependently typed languages ensure correctness prior to execution, while operational semantics define execution without intrinsic guarantees.

We propose a runtime architecture in which each execution step produces a dependent pair:

$$(s', \pi)$$

where s' is the next state and π is a proof that the transition into s' is valid.

The central claim is:

$$\text{execution step} \quad \Longrightarrow \quad \text{state} + \text{proof of validity}.$$

This yields a proof-carrying runtime model for deterministic systems.

The $\lambda_{\Sigma}^{\mathsf{R}}$ Runtime. We refer to the proposed proof-carrying execution model as the $\lambda_{\Sigma}^{\mathsf{R}}$ *runtime*, emphasizing its basis in dependent pair (Σ-type) semantics and its interpretation as a λ-style operational system in which each transition constructs both a next state and a proof of its validity.

We regard $\mathcal{R}$ as an instance of the $\lambda_{\Sigma}^{\mathsf{R}}$ runtime.

Contributions. The main contributions of this work are:

- A formal runtime model, the $\lambda_{\Sigma}^{\mathsf{R}}$ runtime, in which each execution step produces a dependent pair consisting of a next state and a proof of its validity.
- A semantic shift from static verification to proof-carrying execution, where correctness is constructed during runtime rather than solely prior to execution.
- A Gödel-encoded representation of transition rules, enabling runtime transitions to be treated as machine-checkable syntactic objects.

2 Related Work

Dependent type theory, originating in the work of Martin-Löf [1], provides a foundation for interpreting logical propositions as types via the Curry–Howard correspondence [2]. Modern systems such as Coq [5], Lean [4], and Idris [9] realize this paradigm by enabling the construction of programs together with machine-checkable proofs of correctness. In these systems, proof objects are typically constructed at compile time or during interactive proof development, ensuring that only well-typed and verified programs are executed. This approach has been highly successful in formal verification, certified programming, and theorem proving.

Complementary to this, prior work on proof-carrying code [7] and verified runtime systems such as the $\lambda_{\aleph}$ runtime [6] has explored the integration of correctness guarantees with program execution. These approaches either attach proofs to compiled programs or define operational semantics that enforce type safety and effect constraints at runtime. Applied uses of dependent types have been explored in structured input and workflow systems, such as questionnaire flows [10], where types encode constraints on valid interactions. The present work differs from these approaches by introducing a runtime semantics in which each execution step explicitly constructs a dependent pair consisting of both the next state and a proof of transition validity. This shifts proof construction from a purely static artifact to a first-class component of execution.

3 Preliminaries

Let A be a type and let

$$B : A \to \mathcal{U}$$

be a family of types indexed by values of A.

The dependent pair type, or Σ-type, is written:

$$\sum_{x:A} B(x).$$

An element of this type is a pair:

$$(a, b)$$

where:

$$a : A$$

and

$$b : B(a).$$

Thus, a dependent pair contains both a value and evidence that the value satisfies a predicate.

Notation. We write $s, s' \in S$ for states and $\sigma \in \Sigma$ for transition rules. Propositions are written as $P(s, \sigma, s')$, indicating that the transition from s to s' under rule σ is valid. Dependent pair types are written as $\sum_{x:A} B(x)$, and elements are pairs (a, b) where $a : A$ and $b : B(a)$. We write π for proof objects inhabiting such propositions. Gödel encodings of transition rules are written $G(\sigma) \in \mathbb{N}$.

4 Runtime Model

We define a runtime system:

$$\mathcal{R} = (S, \Sigma, T)$$

where:

- S is a set of states;
- Σ is a set of transition rules;
- $T : S \times \Sigma \to S$ is a transition function.

We assume that T is a deterministic function, i.e., for fixed (s, σ), the resulting state s' is uniquely determined.

The ordinary transition function maps a state and a rule to a next state.

However, this ordinary formulation does not itself produce a proof that the transition is valid.

5 Executable Σ-Type Semantics

Definition 1 (Proof-Carrying Transition)**.** *A proof-carrying transition is a function:*

$$\widehat{T} : S \times \Sigma \to \sum_{s':S} P(s, \sigma, s')$$

where:

- $s \in S$ *is the current state;*
- $\sigma \in \Sigma$ *is the transition rule;*
- s' *is the next state;*
- $P(s, \sigma, s')$ *is the proposition that the transition from* s *to* s' *under rule* σ *is valid.*

We assume that P is a decidable or constructively inhabited predicate, and that there exists a computable procedure which, given (s, σ), constructs a proof $\pi : P(s, \sigma, T(s, \sigma))$.

Thus, each execution step produces:

$$(s', \pi)$$

where:

$$\pi : P(s, \sigma, s').$$

The runtime does not merely compute a next state. It computes a next state together with a proof that the transition is admissible.

6 Small-Step Semantics of $\lambda^{\mathsf{R}}_{\Sigma}$

We define a small-step operational semantics in which each transition produces a dependent pair consisting of a next state and a proof of validity.

Definition 2 (Small-Step Judgment)**.** *The small-step relation is written:*

$$(s, \sigma) \;\Rightarrow\; (s', \pi)$$

where:

- $s \in S$ *is the current state;*
- $\sigma \in \Sigma$ *is a transition rule;*
- $s' \in S$ *is the next state;*
- $\pi : P(s, \sigma, s')$ *is a proof that the transition is valid.*

Definition 3 (Transition Rule)**.** *A transition in $\lambda^{\mathsf{R}}_{\Sigma}$ is governed by the rule:*

$$\frac{\sigma \in \Sigma \quad \widehat{T}(s, \sigma) = (s', \pi)}{(s, \sigma) \;\Rightarrow\; (s', \pi)}$$

Definition 4 (Proof-Producing Transition Function)**.** *The transition function is:*

$$\widehat{T}(s, \sigma) = (s', \pi)$$

where $(s', \pi) \in \sum_{s' : S} P(s, \sigma, s')$.

Definition 5 (Multi-Step Execution)**.** *A sequence of transitions is written:*

$$s_0 \;\Rightarrow\; (s_1, \pi_1) \;\Rightarrow\; \cdots \;\Rightarrow\; (s_n, \pi_n).$$

The accumulated proof trace is:

$$(\pi_1, \pi_2, \ldots, \pi_n).$$

7 Gödel Encoding Layer

Let:

$$G : \Sigma \to \mathbb{N}$$

be a Gödel encoding of transition rules.

Each transition rule σ is mapped to a unique natural number:

$$G(\sigma).$$

The proof object may then be represented as:

$$\pi = \text{Valid}(G(\sigma), s, s').$$

This allows transition rules to be treated as arithmetic objects while preserving their syntactic meaning.

8 Comparison with Dependent Type Systems

Dependent type systems construct values of the form:

$$(a, b) \in \sum_{x:A} B(x)$$

at compile time or proof-checking time.

The present runtime model constructs:

$$(s', \pi)$$

during execution.

System	When the Proof Exists
Dependent type system	Before execution
Proof-carrying runtime	During execution

Thus, the proposed architecture shifts proof construction from a purely static layer into the operational behavior of the runtime.

9 Relation to Operational Runtime Semantics

Traditional operational semantics specify how programs step from one state to another.

A typical transition has the form:

$$s \to s'.$$

In the present model, the transition has the form:

$$s \to (s', \pi)$$

where π witnesses the validity of the transition.

This converts ordinary execution into proof-carrying execution.

We use $\to$ for ordinary state transitions and $\Rightarrow$ for proof-carrying transitions in the $\lambda^{\mathsf{R}}_{\Sigma}$ runtime.

10 Minimal Example: Verified Questionnaire Engine

As a concrete illustration—similar to dependent-type applications in structured input systems such as questionnaire flows [10]— we construct a minimal verified flow engine.

Let:

$$S = \{Q1, Q2, Q3, END\}.$$

We interpret inputs such as `Yes`, `No`, and `Text` as transition rules $\sigma \in \Sigma$.

Consider the following transition rules:

$$Q1 + Yes \to Q2$$

$$Q1 + No \to Q3$$

$$Q2 + Text \to END$$

$$Q3 + Text \to END.$$

The proof-carrying transition function returns:

$$\widehat{T}(Q1, Yes) = (Q2, \pi_1)$$

where:

$$\pi_1 : P(Q1, Yes, Q2).$$

Thus the system does not merely advance from $Q1$ to $Q2$. It advances while producing evidence that this transition is valid under the rule system.

11 Properties

Theorem 1 (Certified Execution)**.** *For every valid transition rule σ applied to state s, the runtime produces a dependent pair:*

$$(s', \pi)$$

where:

$$\pi : P(s, \sigma, s').$$

Proof. By definition of $\widehat{T}$, every successful transition returns an element of:

$$\sum_{s':S} P(s, \sigma, s').$$

Therefore, every successful transition includes both a next state and a proof of transition validity. □

Theorem 2 (No Uncertified Reachable States)**.** *Every reachable state produced by the runtime is accompanied by a proof of the transition that produced it.*

Proof. The proof proceeds by induction on the length of execution traces. The base case is the initial state. For the inductive step, assume the current state is certified. A successful transition from that state returns a dependent pair (s', π), where π certifies the transition into s'. Thus s' is certified. □

Theorem 3 (Soundness of $\lambda^{\mathsf{R}}_{\Sigma}$)**.** *For every transition*

$$(s, \sigma) \;\Rightarrow\; (s', \pi)$$

produced by the $\lambda^{\mathsf{R}}_{\Sigma}$ *runtime, the proof object*

$$\pi$$

witnesses the validity of the transition, i.e.,

$$\pi : P(s, \sigma, s').$$

Proof. By definition of the small-step semantics, every transition is derived using the inference rule:

$$\frac{\sigma \in \Sigma \quad \widehat{T}(s, \sigma) = (s', \pi)}{(s, \sigma) \;\Rightarrow\; (s', \pi)}$$

which explicitly requires the existence of a proof π of $P(s, \sigma, s')$.

Thus, every derivable transition preserves the validity predicate P, and no transition can occur without a corresponding proof witness.

Therefore, the system is sound with respect to the validity predicate P. □

[Trace Soundness] Every execution trace produced by the $\lambda^{\mathsf{R}}_{\Sigma}$ runtime consists entirely of certified transitions.

12 Applications

This framework applies naturally to systems where invalid state transitions must be excluded or audited.

Potential applications include:

- verified questionnaire engines;
- insurance underwriting workflows;
- financial compliance systems;
- trading rule validation;
- safety-critical decision systems;
- deterministic failure classification engines.

13 Discussion and Limitations

While the $\lambda^{\mathsf{R}}_{\Sigma}$ runtime provides a principled framework for proof-carrying execution, several limitations remain.

First, the construction of proof objects at runtime introduces computational overhead, particularly in systems where transitions occur at high frequency. In practice, this may require optimization strategies such as proof caching, partial evaluation, or selective proof elision.

Second, the expressiveness of the predicate $P(s, \sigma, s')$ directly impacts the complexity of proof construction. Richer predicates yield stronger guarantees but may require more sophisticated proof generation mechanisms.

Third, the use of Gödel encoding introduces an additional abstraction layer, mapping syntactic objects to arithmetic representations. While this enables uniform treatment of rules, it may obscure the underlying semantics unless carefully managed.

Finally, the present formulation assumes deterministic transitions. Extending the model to nondeterministic or probabilistic systems remains an open direction for future work.

14 Conclusion

We introduced a runtime model in which execution produces proof-carrying state transitions. By interpreting each transition as the construction of a dependent pair, the system bridges dependent type theory, operational semantics, and executable verification.

The resulting architecture suggests a path toward runtimes in which correctness is not only verified before execution, but carried by execution itself.

In future work, we aim to extend the $\lambda^{\mathsf{R}}_{\Sigma}$ runtime to support compositional proof traces and integration with external verification systems.

References

[1] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[2] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry–Howard Isomorphism*. Elsevier, 1998.

[3] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University, 2007.

[4] Leonardo de Moura et al. *The Lean Theorem Prover*. `https://leanprover.github.io/`

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[6] Neal Glew, Tim Sweeney, and Leaf Petersen. *Formalisation of the $\lambda_{\aleph}$ Runtime*. arXiv:1307.5277, 2013.

[7] George Necula. *Proof-Carrying Code*. POPL, 1997.

[8] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica*. 1931.

[9] Edwin Brady. *Idris: A General-Purpose Dependently Typed Programming Language*. Journal of Functional Programming, 2013.

[10] Elisabeth Stenholm. *Dependent Types Are Everywhere!* (Informal talk / exposition), 2025.