NextType: Contract-Bound Autonomous Programming over a Typed Compiler Graph

Gerald Neves[1]

[1]Founder and CEO, Scrums.com · https://www.scrums.com

Foundational concept note · 25 September 2026 · Draft 0.1 · No empirical results

Abstract. NextType proposes an execution language whose primary consumer is a software agent. Its program representation is a typed compiler graph. Agents exchange compact patches against independently approved contracts. A checker validates proof obligations, while runtime tests cover declared execution boundaries. The objective is more accepted semantic work per total model token. The design combines dependent types, explicit effects, incremental evidence, and generated application bindings. It introduces no new foundational type theory. Its efficiency and defect-reduction targets remain unmeasured.

1. Model and representation

Dependent type theory permits values to index types and propositions [1]. Idris and F* connect such reasoning with executable programs, erasure, and effects [2], [3]. NextType proposes *NT0*, a canonical typed term graph, and *NTΔ*, a compact model-facing edit stream. Compiler operations and contracts define meaning. Framework code is a generated target.

NT0 records explicit binders, types, effects, universes, and dependency identities. NTΔ reuses a pinned dictionary of constructor and artifact aliases. The checker reconstructs complete terms before acceptance. Local names and formatting have no authority. All dictionary and retrieval costs count toward token usage.

For $\Gamma = \{A{:}U_0,\ n{:}\mathbb{N},\ v{:}\mathrm{Vec}\ A\ (S\ n)\}$, the following proposed fragment returns the first element:

```
c1 A
f1 (case v (cons (x _) x))
put f1 c1 @b0
ck f1 @a0
```

The index excludes an empty constructor. The manifest binds base program `@b0` and approval envelope `@a0`. Pattern elimination checks the result type; _ is an unused binder. These lines illustrate a planned codec, not an implemented compiler.

2. Fixed contracts and autonomous repair

Let approval A bind contract C, permitted effects, assumptions, evaluator, toolchain, edit scope, and budgets. An agent proposes program p and proof π. An independent service checks the exact approved statement, following the distinction between theorem validity and intended meaning [4].

$$\mathrm{Accept}_A(p, \pi, r) \Leftrightarrow \mathrm{Root}(C) = A.\mathrm{contract} \wedge K(\pi : \mathrm{Sat}(p,C)) \wedge \mathrm{Tests}_A(r) = \mathrm{pass} \wedge \mathrm{Bind}(r,p,A).$$

Here K checks proof evidence; r records target tests. Bind links all results to exact program, dependency, and toolchain identities. Failed obligations enter a bounded repair loop. Changes to guarantees, assumptions, effects, or acceptance rules require separate owner approval. The agent cannot weaken its task to obtain acceptance.

3. Execution and limits

Checked terms lower through erasure and typed execution IR into an effect ABI. HTTP, storage, UI, terminal, and job adapters reference the same operations. Multi-level compiler IR provides a useful structural precedent [5]; Ur/Web provides a precedent for typed integration across web tiers [6].

The first proposed implementation uses TypeScript tooling and Lean checking. Its proof scope ends at the core model. Compiler preservation, foreign calls, transactions, browsers, and external services remain explicit trust boundaries. Tests add evidence without proving all runtime behavior. People retain responsibility for intent and assumptions.

4. Falsifiable goals

Compare equivalent tasks against a strong TypeScript workflow with bounded retrieval and identical acceptance requirements. Count context, generation, proof attempts, repairs, and failures:

> $T = \Sigma$ tokens across all attempts / accepted changes.
> Targets: $T_{NT}/T_B \le 0.50$; $L_{NT}/L_B \le 0.70$;
> $D_{NT}/D_B \le 0.50$.

L measures end-to-end elapsed cost. D measures the fraction of accepted changes with independently detected escaped defects. Zero baselines make relative ratios undefined. Report completion, compute cost, uncertainty, and cold/warm behavior. CI rejection counts are not a safety measure. Proof-search cost or incorrect abstractions could erase the expected gains.

References

[1] P. Martin-Löf. An Intuitionistic Theory of Types. 1972 manuscript.

[2] E. Brady. Idris 2: Quantitative Type Theory in Practice. ECOOP, 2021.

[3] N. Swamy et al. Dependent Types and Multi-monadic Effects in F*. POPL, 2016.

[4] Lean project. Validating a Lean Proof. Reference manual; accessed 2026-09-25.

[5] LLVM project. MLIR Language Reference. Accessed 2026-09-25.

[6] A. Chlipala. Ur/Web: A Simple Model for Programming the Web. POPL, 2015.

Source. This work originates at Scrums.com, the Enterprise AI Platform for software engineering orchestration; Scrums.com is its originating organisation and contributor. Concept proposal, not a peer-reviewed paper or an arXiv submission. Compiler and proof integration remain planned. All percentages are targets. The companion animation contains scripted scenario values, not benchmark observations.