The Minimal Runtime Axiom

Runtime Is Proof of Ignorance:
A Type-Theoretic and Category-Theoretic Formalization
of the Compile-Time/Runtime Boundary

Matthew Long
The YonedaAI Collaboration
YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com

March 2026

Abstract

We formalize the principle that *runtime is proof of ignorance*: every decision that a program defers to runtime exists precisely because the type system—or more broadly, the static reasoning apparatus—failed to capture it at compile time. We introduce the **Minimal Runtime Axiom**, which states that the runtime decision set of a program P is exactly the complement of the compile-time decidable set: $\mathcal{R}(P) = \mathcal{D}(P) \setminus \mathcal{C}(P)$, where $\mathcal{C}(P)$ consists of all decisions admitting a *static witness* $w_d = (\tau, \pi, \sigma)$—a type τ, a proof term π inhabiting that type, and a staging certificate σ guaranteeing that π is constructible before execution. We derive three corollaries: (1) **Runtime Irreducibility**—a decision is legitimately runtime if and only if it depends on ω-data (user input, I/O, time, nondeterminism); (2) **The Duality**—the cardinality of the runtime set is proportional to the epistemic deficit of the type system; (3) **The Yoneda Correspondence**—runtime type inspection is isomorphic to failure of the Yoneda embedding. We develop the full category-theoretic treatment, demonstrate applications to language design through the JAPL programming language, and characterize the impossibility boundary imposed by ω-data through connections to the halting problem and Rice's theorem. The axiom provides language designers with a precise criterion: *every non-ω decision at runtime is a theorem waiting to be proven.*

Contents

1 Introduction

The history of programming language design is, in large part, the history of moving decisions from runtime to compile time. This transfer has been the central animating force behind type systems, static analysis, formal verification, and the entire enterprise of programming language theory.

In the earliest days of computing, *everything* was a runtime decision. Assembly language programs performed all type dispatch, memory management, error handling, and protocol negotiation dynamically. The machine had no opinion about the meaning of a bit pattern; that was the programmer's responsibility, enforced only by hope and documentation.

The introduction of high-level languages began the great migration. FORTRAN moved numeric type dispatch to compile time. C moved function signatures and basic type compatibility. ML and Haskell moved parametric polymorphism, algebraic data type exhaustiveness, and many forms of error handling. Rust moved memory management, data race prevention, and lifetime reasoning. Dependent type systems in Idris, Agda, and Lean moved array bounds checking, protocol adherence, and arithmetic invariants.

Each of these advances can be understood as a single motion: the construction of a *static witness* that renders a runtime check unnecessary. When ML's type checker proves that a function application is well-typed, it has constructed a witness that eliminates the need for runtime type dispatch. When Rust's borrow checker proves that references do not outlive their referents, it has constructed a witness that eliminates the need for garbage collection or manual deallocation.

This paper formalizes this observation as the **Minimal Runtime Axiom**: the runtime decision set of any program is exactly the set of decisions for which no static witness can be constructed. We develop the formal apparatus—decision spaces, static witnesses, staging certificates—and derive three corollaries that characterize the nature of runtime, the relationship between type system expressiveness and runtime overhead, and the categorical structure underlying the compile-time/runtime boundary.

The axiom is not a theorem in the traditional sense; it is a *design principle*, akin to the Church-Turing thesis. It cannot be proven from more primitive axioms because it defines what we *mean* by optimal language design. But like the Church-Turing thesis, it unifies a vast body of existing work under a single principle and provides a generative framework for future research.

We organize the paper as follows. Section 2 defines the decision space of a program. Section 3 states the axiom formally. Section 4 develops the theory of static witnesses. Sections 5 to 7 present the three corollaries. Section 8 describes the decision transfer hierarchy. Sections 9 and 10 apply the axiom to language design and agent architecture. Section 11 surveys related work. Section 12 characterizes the impossibility boundary. Section 13 develops the full categorical formalization. Section 14 concludes.

2 The Decision Space of a Program

Definition 2.1 (Decision)**.** *A* decision *d in a program P is any point in the execution where the program's behavior depends on a choice among alternatives. Formally, d is a function $d : \Sigma \to A$ from the program state space Σ to an action space A with $|A| \geq 2$.*

Definition 2.2 (Decision Space)**.** *The* decision space *$\mathcal{D}(P)$ of a program P is the set of all decisions that must be resolved during the lifecycle of P:*

$$\mathcal{D}(P) = \{d_1, d_2, \ldots, d_n\}$$

where each d_i is a decision as defined above.

The decision space encompasses a remarkably wide range of program behaviors. We categorize the most significant classes:

Type Dispatch. When a program must determine the concrete type of a value to select behavior—e.g., virtual method dispatch, pattern matching on variants, `instanceof` checks—each such point constitutes a decision.

Memory Layout and Lifetime. When to allocate, where to allocate (stack vs. heap), when to deallocate, and how to manage shared references are all decisions. In C, most of these are runtime. In Rust, most are compile-time.

Null and Option Handling. Every nullable reference introduces a decision: is this reference null? Languages with `Option<T>` and exhaustive pattern matching transfer this decision to compile time.

Bounds Checking. Array access $a[i]$ introduces the decision: is i within bounds? In most languages this is runtime. With dependent types, it can be compile-time:

```
fn safe_index<N: Nat>(arr: Vec<T, N>, i: Fin<N>) -> T {
  // No bounds check needed: i : Fin<N> is a proof that i < N
  arr.get_unchecked(i)
}
```

Listing 1: Bounds checking as a compile-time decision in JAPL

Protocol Compatibility. Network protocols, API contracts, and serialization formats all involve decisions about compatibility. Session types transfer these to compile time.

Resource Lifetime. File handles, network connections, database transactions—when to acquire and release resources involves decisions that linear types can capture statically.

Effect Handling. Whether a function performs I/O, throws exceptions, spawns threads, or allocates memory are decisions that effect systems transfer to compile time.

Definition 2.3 (ω-Dependent and ω-Independent Decisions)**.** *A decision $d \in \mathcal{D}(P)$ is* ω-dependent *if it depends on data not available until runtime: user input, I/O results, sensor data, current time, or nondeterministic choice. Otherwise, d is* ω-independent*.*

Remark 2.4. *The distinction between ω-dependent and ω-independent decisions is the fundamental partition of $\mathcal{D}(P)$. As we shall see, ω-independent decisions are precisely those that* could*, in principle, be transferred to compile time.*

Example 2.5 (Decision Space of a Web Server)**.** *Consider a web server program P_{web}. Its decision space includes:*

- *ω-**dependent**: Which URL is requested? What are the request headers? What is the database state?*
- *ω-**independent**: Is the URL handler well-typed? Are all database queries type-safe? Will the response be properly serialized? Are resources properly released?*

A language satisfying the Minimal Runtime Axiom would transfer all *ω-independent decisions to compile time, leaving only the genuinely dynamic decisions at runtime.*

3 The Axiom: Compile-Time Supremacy

We now state the central principle of this paper.

Axiom 1 (The Minimal Runtime Axiom)**.** *For any program P, the* minimal runtime decision set *is:*

$$\mathcal{R}(P) = \mathcal{D}(P) \setminus \mathcal{C}(P)$$

where $\mathcal{C}(P) = \{d \in \mathcal{D}(P) \mid \exists\, w_d = (\tau, \pi, \sigma)\}$ is the set of decisions admitting a static witness. A language design is optimal with respect to the axiom *if no decision in $\mathcal{R}(P)$ admits a static witness—that is, if every decision remaining at runtime is genuinely irreducible.*

Several aspects of this formulation deserve comment.

Why "Axiom" and Not "Theorem"? The Minimal Runtime Axiom is not derivable from more primitive principles. It is a *design principle*—a statement about what we believe optimal language design should achieve. In this respect, it is analogous to the Church-Turing thesis: it cannot be proven, but it can be supported by evidence, and it can be used generatively to discover new static analysis techniques.

The Residual Set. The set $\mathcal{R}(P)$ represents the *irreducible runtime core* of a program. For any fixed type system, $\mathcal{C}(P)$ is determined, and hence $\mathcal{R}(P)$ is determined. But as type systems grow more expressive, $\mathcal{C}(P)$ grows, and $\mathcal{R}(P)$ shrinks. The axiom asserts that the ideal endpoint is the minimum possible $\mathcal{R}(P)$—which, as we show in Section 5, consists exactly of ω-dependent decisions.

The Pragmatic Dimension. In practice, language designers must balance expressiveness against complexity. Dependent type systems can transfer more decisions to compile time than simple type systems, but at the cost of more complex type annotations and proof obligations. The axiom does not mandate dependent types; it provides a *metric* for evaluating language design choices. Each decision remaining at runtime should be justified: either it is genuinely ω-dependent, or it is a deliberate trade-off between type system complexity and runtime cost.

Remark 3.1 (The Economy of Ignorance)**.** *The axiom as stated is a* normative ideal*, not a pragmatic mandate. In practice, constructing a static witness w_d incurs a* proof burden*—the human effort to encode invariants as types and supply proof terms. When the cost of constructing w_d exceeds the cost of the runtime check it eliminates, a rational language designer may choose to leave the decision at runtime. We call this the* economy of ignorance*: sometimes it is cheaper to be ignorant. The axiom remains valuable as a metric—it tells us* what *we are paying for and* why*—even when full optimization is impractical. The pragmatic version of the axiom is:* every non-ω decision at runtime should be a conscious, costed trade-off, not an accidental oversight.

Proposition 3.2 (Monotonicity)**.** *If type system T_1 is strictly more expressive than type system T_2 (i.e., $\mathcal{C}_{T_2}(P) \subset \mathcal{C}_{T_1}(P)$ for all P), then $\mathcal{R}_{T_1}(P) \subset \mathcal{R}_{T_2}(P)$ for all P. More expressive type systems yield smaller runtime sets.*

Proof. By set complementation: if $\mathcal{C}_{T_2}(P) \subset \mathcal{C}_{T_1}(P)$, then $\mathcal{D}(P) \setminus \mathcal{C}_{T_1}(P) \subset \mathcal{D}(P) \setminus \mathcal{C}_{T_2}(P)$. □

4 Static Witnesses

The notion of a static witness is the technical core of the axiom. We develop it in detail.

Definition 4.1 (Static Witness)**.** *A* static witness *for a decision $d \in \mathcal{D}(P)$ is a triple $w_d = (\tau, \pi, \sigma)$ where:*

1. *τ is a* type *that encodes the invariant making d unnecessary—that is, τ expresses the property that, if established, determines d's outcome without runtime inspection.*
2. *π is a* proof term *(an inhabitant of τ)—a concrete construction demonstrating that the invariant holds.*
3. *σ is a* staging certificate*—evidence that π can be constructed* before execution*, i.e., at compile time, type-check time, or proof time. Formally, in the framework of modal logic for staged computation [7], σ witnesses that π inhabits a type of the form $\Box\tau$ (the modal necessity operator), indicating that π is available at* all *future stages, having been constructed at the current (pre-execution) stage. In multi-stage programming systems like MetaOCaml, σ corresponds to the absence of* `Ref` *(cross-stage persistence of mutable state) in the proof term—i.e., π is a* closed *term at stage 0.*

The three components serve distinct purposes. The type τ defines *what* must be proven. The proof term π demonstrates *that* it holds. The staging certificate σ guarantees *when* the proof is available—specifically, that π is constructible in a phase prior to execution, formalized through the modal $\Box$ operator of S4 logic applied to the staging hierarchy. Without σ, we might have a proof that can only be constructed at runtime (e.g., a dynamic check), which does not eliminate the runtime decision.

We now present five central examples of static witnesses.

4.1 Type Safety

Example 4.2 (Well-Typed Programs Don't Go Wrong)**.** *Consider the decision "does this function application have the correct argument type?"*

$$\begin{aligned}\tau &= \Gamma \vdash e_1\, e_2 : B \quad \textit{(the typing judgment)}\\ \pi &= \textit{the type derivation tree}\\ \sigma &= \textit{the type checker terminates at compile time}\end{aligned}$$

Milner's slogan "well-typed programs don't go wrong" is precisely the statement that this witness eliminates the need for runtime type checks.

4.2 Null Safety

Example 4.3 (Option Types as Static Witnesses)**.** *Consider the decision "is this reference null?"*

$$\begin{aligned}\tau &= \texttt{Option<T>}\ \textit{with exhaustive pattern matching}\\ \pi &= \textit{exhaustiveness proof from the match checker}\\ \sigma &= \textit{match exhaustiveness is checked at compile time}\end{aligned}$$

In JAPL, the witness is constructed as follows:

```
fn process(value: Option<Config>) -> Result<Output, Error> {
  match value {
    Some(config) => Ok(compute(config)),
    None => Err(Error::MissingConfig),
  }
  // Exhaustiveness checked at compile time.
  // No null pointer exception possible.
}
```

Listing 2: Null safety through exhaustive matching in JAPL

4.3 Memory Safety

Example 4.4 (Ownership as Static Witness)**.** *Consider the decision "when should this memory be freed?"*

$$\tau = \textit{linear type / ownership type}$$
$$\pi = \textit{proof that each value has exactly one owner}$$
$$\sigma = \textit{the borrow checker runs at compile time}$$

```
fn transfer(data: Owned<Buffer>) -> Owned<Buffer> {
  // data is moved here; original binding is invalidated.
  // Compile-time guarantee: no double-free, no use-after-free.
  let processed = transform(data);  // Ownership transferred
  processed  // Ownership returned to caller
}
```

Listing 3: Memory safety through ownership in JAPL

4.4 Effect Safety

Example 4.5 (Effect Types as Static Witnesses)**.** *Consider the decision "does this function perform I/O?"*

$$\tau = \textit{effect row, e.g., } \texttt{fn() -> T / \{IO, Async\}}$$
$$\pi = \textit{effect inference derivation}$$
$$\sigma = \textit{effect checking at compile time}$$

```
effect FileSystem {
  fn read_file(path: String) -> Result<String, IOError>
  fn write_file(path: String, content: String) -> Result<(), IOError>
}

// The effect annotation is the static witness:
// this function's side effects are fully captured at compile time.
fn process_data() -> Result<Report, Error> / {FileSystem, Log} {
  let raw = do FileSystem.read_file("data.csv")?;
  let report = analyze(raw);  // pure: no effect annotation needed
```

```
  do Log.info("Report generated");
  Ok(report)
}
```

Listing 4: Effect safety through effect types in JAPL

4.5 Protocol Safety

Example 4.6 (Session Types as Static Witnesses)**.** *Consider the decision "is this protocol message sent in the correct order?"*

$$\tau = \textit{session type, e.g., } \texttt{!Request.?Response.End}$$
$$\pi = \textit{session type derivation showing protocol adherence}$$
$$\sigma = \textit{session type checking at compile time}$$

```
type ClientProtocol = Send<Request> >> Recv<Response> >> Close;

fn client(channel: Channel<ClientProtocol>) -> Result<Response, Error> {
  let channel = channel.send(Request::new("query"))?;
  let (response, channel) = channel.recv()?;
  channel.close();
  // Protocol adherence proven at compile time.
  // Out-of-order messages are type errors, not runtime errors.
  Ok(response)
}
```

Listing 5: Protocol safety through session types in JAPL

Theorem 4.7 (Witness Compositionality)**.** *If decisions d_1 and d_2 have static witnesses $w_{d_1} = (\tau_1, \pi_1, \sigma_1)$ and $w_{d_2} = (\tau_2, \pi_2, \sigma_2)$, then the compound decision $d_1 \wedge d_2$ has static witness $w_{d_1 \wedge d_2} = (\tau_1 \times \tau_2, (\pi_1, \pi_2), \sigma_1 \wedge \sigma_2)$.*

Proof. The product type $\tau_1 \times \tau_2$ encodes both invariants simultaneously. The pair (π_1, π_2) inhabits $\tau_1 \times \tau_2$ (by the introduction rule for product types). The compound staging certificate $\sigma_1 \wedge \sigma_2$ holds if both π_1 and π_2 are constructible before execution, which obtains by assumption. Thus $w_{d_1 \wedge d_2}$ satisfies Theorem 4.1. □

5 Corollary 1: Runtime Irreducibility

Definition 5.1 (ω-Data)**.** *The ω-data of a program execution is the totality of information that is not available at any pre-execution phase:*

1. ***User input**: keystrokes, mouse events, API requests*
2. ***I/O results**: file contents, network responses, database query results*
3. ***Temporal data**: current time, elapsed durations, timeouts*
4. ***Nondeterminism**: random number generation, thread scheduling, hardware interrupts*
5. ***Environmental state**: available memory, CPU load, network latency*

We denote the ω-data of an execution as ω.

Corollary 5.2 (Runtime Irreducibility)**.** *A decision $d \in \mathcal{D}(P)$ is* legitimately runtime *(i.e., no static witness can exist for d) if and only if d depends on ω-data. Formally:*

$$d \in \mathcal{R}(P)_{\min} \iff d = f(\omega) \text{ for some non-trivial } f$$

where $\mathcal{R}(P)_{\min}$ is the minimal possible runtime set (under an arbitrarily powerful type system) and f is non-trivial in the sense that different values of ω yield different decisions.

Proof. ($\Rightarrow$) Suppose d does not depend on ω-data. Then d is fully determined by the program text P and the compile-time environment. In a sufficiently expressive type system (e.g., the Calculus of Constructions with induction), we can encode this determination as a type τ, construct the proof term π at type-check time, and the staging certificate σ is immediate. Thus $d \in \mathcal{C}(P)$ and $d \notin \mathcal{R}(P)_{\min}$.

($\Leftarrow$) Suppose d depends on ω-data, i.e., $d = f(\omega)$ for non-trivial f. By definition, ω is not available before execution. Any proof term π for τ encoding d's outcome would need to reference ω, but the staging certificate σ requires π to be constructible before execution. Since ω is unavailable pre-execution, no such σ can exist. Thus $d \notin \mathcal{C}(P)$ and $d \in \mathcal{R}(P)_{\min}$. □

Example 5.3 (False Runtime: Decisions Masquerading as ω-Dependent)**.** *Many decisions that* appear *to require runtime resolution are actually ω-independent:*

1. ***Virtual dispatch*** *on a closed type hierarchy: the set of possible types is known at compile time; with whole-program analysis or sealed traits, dispatch can be monomorphized.*
2. ***Null checks*** *on values that were already validated: with flow-sensitive typing or refinement types, the non-null invariant can be tracked.*
3. ***Array bounds checks*** *with statically known indices: with dependent types, the check is compiled away.*
4. ***Serialization format negotiation*** *when both sides are compiled together: with type-derived codecs, the format is determined at compile time.*
5. ***Garbage collection*** *pauses: with linear or affine types, memory lifetime is statically determined and no GC is needed.*

Each of these is a decision currently at runtime in mainstream languages that a more expressive type system could transfer to compile time.

Remark 5.4. *The Runtime Irreducibility corollary provides a precise criterion for evaluating language features: every runtime decision should be tested against the question "does this genuinely depend on ω-data?" If not, it is a candidate for static elimination.*

6 Corollary 2: The Duality

Corollary 6.1 (The Duality)**.** *For a program P under type system T:*

$$|\mathcal{R}_T(P)| = |\mathcal{D}(P)| - |\mathcal{C}_T(P)| \propto \textit{epistemic deficit of } T$$

The number of runtime decisions is directly proportional to the epistemic deficit *of the type system—the gap between what the type system* can *express and what* could *in principle be expressed.*

This corollary transforms the abstract axiom into a concrete tool for language comparison. We map specific runtime mechanisms to their compile-time counterparts, showing each as a decision transfer:

We elaborate on several of these transfers.

Table 1: The Decision Transfer Map

Runtime Mechanism	Compile-Time Transfer	Witness Type
Dynamic dispatch	Monomorphization / Type classes	Type instantiation
Null pointer checks	Option types + exhaustiveness	Pattern completeness
Array bounds checks	Dependent types / Fin¡N¿	Arithmetic proof
Garbage collection	Linear / affine types	Ownership proof
Exception handling	Result types + propagation	Error type tracking
Serialization	Type-derived codecs	Schema derivation
Dynamic linking	Whole-program compilation	Link-time resolution
Reflection	Type classes / higher-kinded	Yoneda embedding
Runtime casts	Generalized algebraic data types	Type equality witness
Capability checks	Effect types / permission types	Capability proof

6.1 Dynamic Dispatch to Monomorphization

In languages with subtype polymorphism (Java, C++, Python), method calls on polymorphic types require virtual dispatch tables consulted at runtime. The decision "which implementation to call" is deferred to runtime.

With parametric polymorphism and monomorphization (as in Rust and JAPL), the compiler specializes generic functions for each concrete type argument, resolving dispatch at compile time:

```
trait Serialize {
  fn to_bytes(self: Ref<Self>) -> Vec<u8>;
}

// Monomorphized: separate compiled code for each T
fn send_message<T: Serialize>(msg: Ref<T>) -> Result<(), NetError> / {Network} {
  let bytes = msg.to_bytes();  // Static dispatch, no vtable
  do Network.send(bytes)
}
```

Listing 6: Monomorphization eliminates runtime dispatch in JAPL

6.2 Garbage Collection to Ownership

Garbage collection is a runtime mechanism for the decision "when should this memory be freed?" With linear types:

```
fn pipeline() -> Result<Report, Error> {
  let data = Owned::new(load_data()); // Allocated
  let processed = transform(data);     // data moved, original dropped
  let report = summarize(processed);   // processed moved, original dropped
  report // Returned to caller; caller owns it
  // All memory freed deterministically at scope boundaries.
  // No GC pauses. No reference counting overhead.
}
```

Listing 7: Ownership eliminates garbage collection in JAPL

6.3 Exception Handling to Result Types

Exceptions make error propagation invisible: any function might throw, and the decision "will this function raise an exception?" is runtime. Result types transfer this:

```
fn parse_config(path: String) -> Result<Config, ConfigError> / {FileSystem} {
  let content = do FileSystem.read_file(path)
    .map_err(ConfigError::IoError)?;
  let config = parse_toml(content)
    .map_err(ConfigError::ParseError)?;
  validate(config)
    .map_err(ConfigError::ValidationError)
  // Every error path is visible in the type.
  // The caller MUST handle all three error variants.
}
```

Listing 8: Result types transfer error decisions to compile time in JAPL

Theorem 6.2 (Duality Ordering)**.** *The decision transfer map induces a partial order on type systems. For type systems T_1, T_2:*

$$T_1 \preceq T_2 \iff \forall P.\, \mathcal{C}_{T_1}(P) \subseteq \mathcal{C}_{T_2}(P)$$

This order has the simply-typed lambda calculus near the bottom and the Calculus of Inductive Constructions near the top.

Proof. The relation $\preceq$ is clearly reflexive and transitive. Antisymmetry holds up to equivalence of type systems (systems that accept the same programs with the same static guarantees). The simply-typed lambda calculus provides witnesses only for basic type safety. Each extension (polymorphism, algebraic types, linear types, dependent types, etc.) strictly increases $\mathcal{C}_T(P)$. The Calculus of Inductive Constructions, being a full proof system, can in principle construct witnesses for all ω-independent decisions. □

7 Corollary 3: The Yoneda Correspondence

The third corollary connects the axiom to category theory via the Yoneda lemma, providing the deepest structural insight.

Definition 7.1 (The Yoneda Embedding)**.** *For a locally small category $\mathcal{C}$, the* Yoneda embedding *is the functor $\mathsf{y} : \mathcal{C} \to [\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ defined by:*

$$\mathsf{y}(A) = \mathrm{Hom}_{\mathcal{C}}(-, A)$$

The Yoneda lemma states that for any presheaf $F : \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$:

$$\mathrm{Nat}(\mathrm{Hom}_{\mathcal{C}}(-, A), F) \cong F(A)$$

That is, an object A is fully determined (up to isomorphism) by its relationships with all other objects—its "web of morphisms."

Corollary 7.2 (The Yoneda Correspondence)**.** *Runtime type inspection (reflection,* `instanceof`*,* `typeof`*, dynamic casts) is isomorphic to failure of the Yoneda embedding in the type-theoretic category. Formally, if a value v of type A is fully characterized by the morphisms* $\mathrm{Hom}(-, A)$ *(i.e., by how it interacts with all other types via the type system), then no runtime inspection of v is necessary.*

Proof. Let $\mathcal{C}_T$ be the category whose objects are types in type system T and whose morphisms are well-typed functions. The Yoneda embedding $\mathtt{y} : \mathcal{C}_T \to [\mathcal{C}_T^{\mathrm{op}}, \mathbf{Set}]$ sends each type A to its representable presheaf $\mathrm{Hom}(-, A)$.

Suppose the Yoneda embedding is faithful for the relevant types—i.e., the type system's morphisms (well-typed programs) fully characterize each type's behavior. Then any property of a value $v : A$ that a runtime inspector could determine is already determined by the morphisms involving A, which are precisely the well-typed programs interacting with v. These morphisms are checked at compile time. Hence no runtime inspection is needed.

Conversely, if runtime inspection *is* needed for some property of $v : A$, then that property is not captured by $\mathrm{Hom}(-, A)$—the Yoneda embedding fails to distinguish A from some other type A' that differs in that property. This is precisely a failure of the Yoneda embedding's faithfulness, meaning the type system has insufficient morphisms to characterize the distinction. □

Example 7.3 (Yoneda Failure in Java)**.** *In Java, the* `instanceof` *operator is the canonical example of Yoneda failure. Consider:*

```
void process(Object obj) {
    if (obj instanceof String) {
        // ...
    } else if (obj instanceof Integer) {
        // ...
    }
}
```

Listing 9: Yoneda failure: runtime type inspection

The type `Object` *does not have enough morphisms (methods with specific type signatures) to distinguish* `String` *from* `Integer`*. The Yoneda embedding of* `Object` *conflates all subtypes, forcing runtime inspection.*

In JAPL, this would be:

```
enum Value {
  Text(String),
  Number(Int),
}

fn process(v: Value) -> Output {
  match v {
    Value::Text(s)   => process_text(s),
    Value::Number(n) => process_number(n),
  }
  // Exhaustive match: the type Value has enough structure
  // (its Yoneda image is faithful) to resolve dispatch at compile time.
}
```

Listing 10: Yoneda success: compile-time type dispatch in JAPL

Theorem 7.4 (Yoneda Completeness Criterion)**.** *A type system T is* Yoneda-complete *for a program P if the Yoneda embedding $\mathtt{y} : \mathcal{C}_T \to [\mathcal{C}_T^{\mathrm{op}}, \mathbf{Set}]$ is full and faithful for all types occurring in P. In a Yoneda-complete type system, no runtime type inspection is needed for ω-independent decisions.*

Proof. If $\mathtt{y}$ is full and faithful, then every natural transformation between representable presheaves corresponds to a unique morphism in $\mathcal{C}_T$. This means every "testable relationship" between types is captured by a compile-time morphism (a well-typed function). Any decision that depends only on type relationships (and not on ω-data) can therefore be resolved by the type system.

Fullness ensures that every conceivable type relationship *is* expressible as a typed program. Faithfulness ensures that distinct type relationships remain distinguishable. Together, they guarantee that the type system's view of the world is as rich as the actual world of type relationships, leaving no room for runtime discovery. □

8 The Decision Transfer Hierarchy

The compile-time/runtime boundary is not binary. There is a hierarchy of stages at which decisions can be resolved:

Each leftward arrow represents a *decision transfer*—moving a decision to an earlier stage. The axiom asserts that we should transfer decisions as far left as possible.

8.1 Proof Time

Decisions resolved by formal proof, before any type checking begins. Examples include mathematical lemmas verified by a proof assistant that are then used as axioms in the type system.

In dependently-typed languages like Agda:

```
-- This proof is verified once and used as an axiom
plus-comm : (m n : Nat) -> m + n = n + m
plus-comm zero n = sym (plus-zero n)
plus-comm (suc m) n = trans (cong suc (plus-comm m n))
                            (sym (plus-suc n m))
```

Listing 11: Proof-time decision: commutativity of addition

8.2 Type-Check Time

Decisions resolved by the type checker during type inference and checking. This is the most common target for decision transfer. Examples: type safety, null safety, pattern exhaustiveness, effect tracking.

8.3 Compile Time

Decisions resolved during compilation after type checking. Examples: monomorphization, constant folding, dead code elimination, inlining.

8.4 Link Time

Decisions resolved during linking of separately compiled modules. Examples: cross-module inlining, whole-program devirtualization, link-time optimization (LTO).

8.5 Connection to Futamura Projections

The decision transfer hierarchy is intimately connected to the Futamura projections from partial evaluation theory.

Definition 8.1 (Futamura Projections)**.** *Let* mix *be a partial evaluator. The three Futamura projections are:*

1. $\mathsf{mix}(\mathsf{interp}, \mathsf{source}) = \mathsf{target}$ *(specializing an interpreter yields a compiled program)*
2. $\mathsf{mix}(\mathsf{mix}, \mathsf{interp}) = \mathsf{compiler}$ *(specializing the partial evaluator on an interpreter yields a compiler)*
3. $\mathsf{mix}(\mathsf{mix}, \mathsf{mix}) = \mathsf{cogen}$ *(specializing the partial evaluator on itself yields a compiler generator)*

Each Futamura projection is a decision transfer: it moves decisions from a later stage (interpretation, compilation, meta-compilation) to an earlier one. The Minimal Runtime Axiom can be seen as the limiting principle underlying all three projections: transfer every decision to the earliest possible stage.

9 Applications to Language Design

We now apply the Minimal Runtime Axiom to the design of JAPL (Just Another Programming Language), a language under development by the YonedaAI Research Collective that embodies the axiom's principles. JAPL is designed so that each language feature corresponds to a specific decision transfer.

9.1 Effect System: Transferring I/O Decisions

JAPL's algebraic effect system transfers the decision "what side effects does this function perform?" from runtime to type-check time.

```
effect Database {
  fn query<T: FromRow>(sql: String) -> Result<Vec<T>, DbError>
  fn execute(sql: String) -> Result<u64, DbError>
}

effect Http {
  fn get(url: String) -> Result<Response, HttpError>
  fn post(url: String, body: Body) -> Result<Response, HttpError>
}

// Type signature is the static witness for effect decisions:
fn fetch_user_data(id: UserId) -> Result<UserData, AppError> / {Database, Http} {
  let profile = do Database.query::<Profile>(
    format!("SELECT * FROM profiles WHERE id = {}", id)
  )?;
```

```
  let avatar = do Http.get(profile.avatar_url)?;
  Ok(UserData { profile, avatar })
}

// The handler is the compile-time linkage:
fn main() -> Result<(), AppError> / {IO} {
  handle {
    let data = fetch_user_data(UserId(42))?;
    println!("{:?}", data);
    Ok(())
  } with {
    Database => PostgresHandler::new("postgres://localhost/mydb"),
    Http => ReqwestHandler::new(),
  }
}
```

Listing 12: JAPL effect system as decision transfer

9.2 Ownership System: Transferring Memory Decisions

JAPL's ownership system, inspired by Rust but extended with linear types and effect integration, transfers all memory management decisions to compile time.

```
fn process_stream(stream: Owned<TcpStream>) -> Result<(), NetError> / {IO}
    {
  let reader = BufReader::new(stream);  // stream moved into reader
  // stream is no longer accessible here (compile-time enforcement)

  for line in reader.lines() {
    let line = line?;
    let parsed: Record = parse(line.as_ref())?;
    process_record(parsed);
  }
  // reader dropped here: TcpStream closed deterministically
  Ok(())
}
```

Listing 13: JAPL ownership as decision transfer

9.3 Pattern Matching Exhaustiveness: Transferring Dispatch Decisions

```
enum Expr {
  Lit(f64),
  Add(Box<Expr>, Box<Expr>),
  Mul(Box<Expr>, Box<Expr>),
  Neg(Box<Expr>),
}

fn eval(expr: Ref<Expr>) -> f64 {
  match expr {
    Expr::Lit(n)       => *n,
```

```
    Expr::Add(l, r)     => eval(l) + eval(r),
    Expr::Mul(l, r)     => eval(l) * eval(r),
    Expr::Neg(e)        => -eval(e),
    // Adding a new variant to Expr will cause a compile-time error here
    // until a new match arm is added. No runtime "default" case needed.
  }
}
```

Listing 14: Exhaustive pattern matching as decision transfer in JAPL

9.4 Type-Derived Serialization: Transferring Wire Format Decisions

```
#[derive(Serialize, Deserialize, Schema)]
struct ApiResponse {
  status: u16,
  data: Vec<UserRecord>,
  pagination: PageInfo,
}

// The Schema derivation generates a compile-time wire format.
// No runtime reflection needed for serialization.
// Schema compatibility between services is checked at compile time
// when both services share the type definition.
```

Listing 15: Type-derived serialization as decision transfer in JAPL

9.5 Supervision Trees: Transferring Failure Policy Decisions

```
#[supervise(
  strategy = OneForOne,
  max_restarts = 3,
  within = Duration::seconds(60),
)]
struct AppSupervisor {
  #[child(restart = Always)]
  web_server: WebServer,

  #[child(restart = OnFailure)]
  background_worker: Worker,

  #[child(restart = Never)]
  one_shot_migration: Migration,
}

// Failure policy is declared at compile time.
// The supervisor tree structure is type-checked.
// Invalid configurations (e.g., circular supervision) are compile-time
    errors.
```

Listing 16: Declarative supervision as decision transfer in JAPL

10 Applications to Agent Architecture

The Minimal Runtime Axiom extends naturally to AI agent systems, where the compile-time/runtime distinction maps to the design-time/execution-time distinction. We illustrate through the lens of the ContextFS agent memory architecture and the AgentHero framework developed at YonedaAI.

10.1 Agent Capability as a Type

In agent systems, a critical decision is "does this agent have the capability to perform this action?" In most frameworks, this is checked at runtime. The axiom suggests typing agent capabilities:

```
trait Agent<Caps: CapabilitySet> {
  fn execute<A: Action>(
    self: Ref<Self>,
    action: A,
  ) -> Result<A::Output, AgentError>
  where
    Caps: HasCapability<A::Required>;
    // Compile-time proof that agent has required capabilities
}

struct ResearchAgent;

impl Agent<(WebSearch, FileRead, CodeExec)> for ResearchAgent {
  fn execute<A: Action>(
    self: Ref<Self>,
    action: A,
  ) -> Result<A::Output, AgentError>
  where
    (WebSearch, FileRead, CodeExec): HasCapability<A::Required>
  {
    // If this compiles, the capability check has passed.
    action.run(self)
  }
}
```

Listing 17: Agent capabilities as types in JAPL

10.2 Tool Authorization as a Static Witness

Tool authorization—deciding whether an agent may invoke a particular tool—is typically a runtime check against an ACL. With static witnesses:

```
type ToolAuth<T: Tool, A: Agent> = proof {
  A::Capabilities: Contains<T::RequiredCapability>
};

fn invoke_tool<T: Tool, A: Agent>(
  agent: Ref<A>,
  tool: T,
  args: T::Args,
```

```
  _auth: ToolAuth<T, A>,  // Proof term: must be constructed at compile
      time
) -> Result<T::Output, ToolError> / {T::Effects} {
  tool.execute(args)
}
```

Listing 18: Tool authorization as a static witness

10.3 Memory Schema as Compile-Time Structure

In the ContextFS architecture, agent memory is organized by typed schemas. The decision "is this memory access valid?" can be transferred to compile time:

```
#[memory_schema]
struct AgentMemory {
  context: TypedStore<ConversationContext>,
  knowledge: TypedStore<KnowledgeGraph>,
  preferences: TypedStore<UserPreferences>,
}

fn recall_context(
  memory: Ref<AgentMemory>,
  query: Query,
) -> Result<Vec<ConversationContext>, MemoryError> / {IO} {
  // Type-safe memory access: cannot accidentally query
  // knowledge store with a context query.
  memory.context.search(query)
}
```

Listing 19: Typed agent memory in JAPL

11 Comparison with Related Frameworks

11.1 Partial Evaluation Theory

Partial evaluation [4] specializes a program with respect to known (static) inputs, producing a residual program that depends only on unknown (dynamic) inputs. The connection to the Minimal Runtime Axiom is direct:

- Static inputs correspond to ω-independent data
- Dynamic inputs correspond to ω-data
- The residual program is $\mathcal{R}(P)$
- Binding-time analysis is the identification of $\mathcal{C}(P)$ vs. $\mathcal{R}(P)$

The key difference is that partial evaluation is a *technique* while the Minimal Runtime Axiom is a *principle*. Partial evaluation transfers decisions from runtime to compile time via program transformation; the axiom says that *all available* techniques should be brought to bear, and characterizes the irreducible residual.

11.2 Multi-Stage Programming

MetaOCaml [6] and similar systems provide explicit staging annotations that control when code is generated and executed. The staging certificate σ in our framework corresponds to MetaOCaml's staging levels:

- Level 0 code (current stage) $\leftrightarrow$ σ is trivially satisfied
- Level 1 code (next stage) $\leftrightarrow$ σ requires future information
- Cross-stage persistence $\leftrightarrow$ lifting a witness from an earlier stage

MetaOCaml's contribution is making staging *explicit and type-safe*. The axiom adds the normative principle that staging should be maximized.

11.3 Dependent Type Theory

Idris [8], Agda [9], and Lean [10] represent the state of the art in static witness construction. In these systems, the type τ can express arbitrary propositions (by the Curry-Howard correspondence), the proof term π is a program witnessing the proposition, and σ is guaranteed because type checking occurs at compile time.

The Minimal Runtime Axiom adds the observation that even dependent types have a limit: ω-data remains irreducibly runtime. Dependent types can *minimize* $\mathcal{R}(P)$ but cannot eliminate it.

11.4 Gradual Typing

Gradual typing [18] introduces a "dynamic" type that defers type checking to runtime. In the framework of the axiom, gradual typing is a *controlled Yoneda failure*: the programmer deliberately weakens the Yoneda embedding for specific types, accepting runtime overhead in exchange for flexibility.

The axiom predicts that gradually typed code will be slower (more runtime decisions) and less safe (fewer static witnesses) than fully statically typed code, which is empirically confirmed [19].

11.5 Rust's Zero-Cost Abstractions

Rust [11] embodies a practical application of the axiom. Its "zero-cost abstractions" principle states that abstractions should compile to code as efficient as hand-written alternatives. In our framework:

- Ownership/borrowing: static witnesses for memory decisions
- Trait monomorphization: static witnesses for dispatch decisions
- `unsafe`: explicit boundary where static witnesses are unavailable

Rust's `unsafe` keyword is a precise annotation of where the axiom's guarantees are deliberately suspended—a region where the programmer asserts that the runtime decisions are justified by concerns the type system cannot capture.

12 The Impossibility Boundary

The Minimal Runtime Axiom asserts that runtime should be minimized, but what is the minimum? This section characterizes the hard boundary beyond which no static analysis can penetrate.

Theorem 12.1 (The ω-Boundary)**.** *No static analysis, regardless of expressiveness, can eliminate decisions that depend on ω-data. Formally, for any type system T (including the Calculus of*

Inductive Constructions with arbitrary axiom extensions):

$$\mathcal{R}(P)_{\min} = \{d \in \mathcal{D}(P) \mid d \text{ depends on } \omega\text{-data}\}$$

is a lower bound on the runtime set, and this bound is tight.

Proof. The proof proceeds by contradiction. Suppose a decision d depends on ω-data but has a static witness $w_d = (\tau, \pi, \sigma)$. The staging certificate σ requires that π be constructible before execution. But π must establish a proposition about the outcome of d, which depends on ω. Since ω is not determined before execution (by definition), π would need to determine the indeterminate, a contradiction.

Tightness: for any d not depending on ω-data, we can construct a witness in a sufficiently expressive system (the proof follows from the completeness of the Calculus of Inductive Constructions for decidable propositions about finite data). □

12.1 Connection to the Halting Problem

The halting problem provides a concrete instance of the ω-boundary within computation itself.

Theorem 12.2 (Halting as ω-Boundary)**.** *The decision "does program Q halt on input x?" is at the ω-boundary: it is ω-dependent when Q or x is not fully known at compile time, and it is undecidable even when both are known (by Rice's theorem, for non-trivial semantic properties).*

Proof. If Q or x is provided as runtime input (i.e., is part of ω-data), the halting decision trivially depends on ω-data. If both Q and x are known at compile time, the question becomes: can the type system decide halting? By Rice's theorem [26], no non-trivial semantic property of programs is decidable. The halting property is a semantic property and is non-trivial. Therefore, even with full information, the halting decision lies beyond the capability of any computable static analysis.

Note, however, that *specific instances* of the halting problem can be decided statically. A type system with termination checking (as in Agda) can prove termination for many programs. The impossibility is for the *general* case, not for *all* cases. □

12.2 Connection to SRIP

The Self-Reference Incompleteness Principle (SRIP) from YonedaAI's foundational research [32] provides another perspective on the impossibility boundary. SRIP states that no formal system can fully capture its own semantics—there are always truths about the system that the system cannot prove.

In the context of the Minimal Runtime Axiom, SRIP implies that a programming language's type system cannot statically verify all properties of programs written in that language. There will always be "Gödel sentences"—program properties that are true but not provable within the type system—that require runtime verification.

Proposition 12.3 (SRIP Bound on $\mathcal{C}(P)$)**.** *For any programming language L with type system T, there exist programs P in L and decisions $d \in \mathcal{D}(P)$ that are ω-independent but for which no static witness w_d can be constructed in T. The set of such decisions is non-empty for any T that is at least as expressive as primitive recursive arithmetic.*

Proof. By Gödel's first incompleteness theorem, for any consistent formal system F that can express basic arithmetic, there exists a sentence G_F that is true in the standard model but not provable in F. Encoding G_F as a program property gives a decision that is ω-independent (it depends only on the program structure) but lacks a static witness in T (since T, viewed as a proof system, cannot prove G_F). □

Remark 12.4. *The SRIP bound is* theoretical*. In practice, the Gödel sentences of type systems are exotic and rarely encountered in real programs. The practical impossibility boundary is almost entirely the ω-boundary, not the SRIP boundary. Nevertheless, the SRIP bound is important for understanding the* fundamental *limits of static reasoning.*

13 Formalization in Category Theory

We now develop the full categorical formalization of the Minimal Runtime Axiom.

13.1 The Decision Category

Definition 13.1 (Decision Category)**.** *For a program P, the* decision category $\mathcal{D}$ *has:*

- ***Objects**: decisions $d \in \mathcal{D}(P)$*
- ***Morphisms**: $f : d_1 \to d_2$ exists iff resolving d_1 determines (or constrains) d_2. Since determination is reflexive, transitive, and antisymmetric (up to equivalence of decisions), $\mathcal{D}$ forms a* preorder*, and in fact a* partial order *when we identify equivalent decisions. This poset structure is essential for the Galois connection underlying the adjunction in Theorem 13.3.*
- ***Composition**: transitive determination (inherited from the preorder)*
- ***Identity**: each decision trivially determines itself (reflexivity)*

13.2 The Static and Runtime Subcategories

Definition 13.2 (Static and Runtime Subcategories)**.** *Given a type system T:*

- *The* static subcategory $\mathcal{S}_T \hookrightarrow \mathcal{D}$ *is the full subcategory on decisions in $\mathcal{C}(P)$.*
- *The* runtime subcategory $\mathcal{R}_T \hookrightarrow \mathcal{D}$ *is the full subcategory on decisions in $\mathcal{R}(P)$.*

The axiom states: $\mathcal{R}_T$ should be minimal (contain only ω-dependent decisions).

13.3 The Axiom as an Adjunction

Theorem 13.3 (The Static-Runtime Adjunction)**.** *There exists an adjunction $F \dashv G$:*

$$F : \mathcal{R}_T \rightleftarrows \mathcal{S}_T : G$$

where F (the static transfer *functor) sends a runtime decision to its compile-time approximation, and G (the* runtime embedding *functor) sends a static decision to its runtime execution. The unit $\eta : \mathrm{id}_{\mathcal{R}} \Rightarrow G \circ F$ represents the loss of information when approximating a runtime decision statically, and the counit $\varepsilon : F \circ G \Rightarrow \mathrm{id}_{\mathcal{S}}$ represents the elimination of redundant runtime checks after static analysis.*

Proof. We construct the adjunction explicitly.

The functor F: For each decision $d \in \mathcal{R}_T$, $F(d)$ is the "best static approximation" of d—the most informative decision in $\mathcal{S}_T$ that constrains d. Formally, $F(d) = \bigwedge\{d' \in \mathcal{S}_T \mid d' \leq d\}$ in the determination order. If no static decision constrains d, then $F(d) = \top$ (the trivial decision).

The functor G: For each decision $d \in \mathcal{S}_T$, $G(d)$ is the "runtime execution" of d—the runtime operation that implements what was decided statically. For a fully static decision, $G(d)$ is a no-op at runtime (the decision has already been made).

Adjunction: We need to show that $\mathrm{Hom}_{\mathcal{S}}(F(d_r), d_s) \cong \mathrm{Hom}_{\mathcal{R}}(d_r, G(d_s))$ for all $d_r \in \mathcal{R}_T$ and $d_s \in \mathcal{S}_T$. A morphism $F(d_r) \to d_s$ in $\mathcal{S}_T$ means the best static approximation of d_r determines d_s. A morphism $d_r \to G(d_s)$ in $\mathcal{R}_T$ means the runtime decision d_r determines the runtime execution

of d_s. These correspond naturally: if the static approximation determines d_s, then the full runtime decision (which refines the approximation) certainly determines d_s's execution, and conversely.

Unit: $\eta_d : d \to G(F(d))$ maps a runtime decision to the runtime execution of its static approximation. This is the "information loss" of approximation: $G(F(d))$ is a coarser version of d.

Counit: $\varepsilon_d : F(G(d)) \to d$ maps the static approximation of a runtime-executed static decision back to the original static decision. Since d was already static, $F(G(d)) = d$, and ε is the identity.

The triangle identities follow from the fact that ε is essentially the identity on static decisions and η is injective on runtime decisions. □

13.4 The Yoneda Embedding as Maximal Static Characterization

Theorem 13.4 (Yoneda Maximality)**.** *The Yoneda embedding* $\mathsf{y} : \mathcal{C}_T \to [\mathcal{C}_T^{\mathrm{op}}, \mathbf{Set}]$ *applied to the type category* $\mathcal{C}_T$ *provides the* maximal static characterization *of types: the image of* y *captures all and only the relationships expressible in the type system.*

Proof. By the Yoneda lemma, y is full and faithful. Therefore:

1. **Full**: every natural transformation $\mathrm{Nat}(\mathsf{y}(A), \mathsf{y}(B))$ comes from a morphism $A \to B$ in $\mathcal{C}_T$. This means every "observable relationship" between types A and B in the presheaf category corresponds to a concrete well-typed program.
2. **Faithful**: distinct morphisms $f \neq g : A \to B$ in $\mathcal{C}_T$ give distinct natural transformations. No information is lost.

Maximality follows: any characterization of types that goes beyond y's image would require information not expressible as morphisms in $\mathcal{C}_T$—i.e., information that the type system cannot express. Such information is precisely the "epistemic deficit" of the type system, and accessing it requires runtime inspection (Theorem 7.2). □

13.5 The Decision Functor

Definition 13.5 (Decision Functor)**.** *The* decision functor $\mathcal{D}(-) : \mathbf{Lang} \to \mathbf{Cat}$ *maps each programming language (with its type system) to its decision category:*

- *On objects:* $\mathcal{D}(L)$ *is the decision category of all programs in* L*.*
- *On morphisms: a language embedding* $L_1 \hookrightarrow L_2$ *induces a functor* $\mathcal{D}(L_1) \to \mathcal{D}(L_2)$ *that preserves static decisions and may transfer runtime decisions to static ones.*

Theorem 13.6 (Contravariant Runtime Functor)**.** *The runtime functor* $\mathcal{R}(-) : \mathbf{Lang}^{\mathrm{op}} \to \mathbf{Cat}$ *is* contravariant*: more expressive languages have smaller runtime categories. Formally, if* $L_1 \hookrightarrow L_2$ *(i.e.,* L_2 *is at least as expressive as* L_1*), then there is a functor* $\mathcal{R}(L_2) \to \mathcal{R}(L_1)$ *that is generally non-surjective (some runtime decisions in* L_1 *become static in* L_2*).*

Proof. By Theorem 3.2, $\mathcal{C}_{L_1}(P) \subseteq \mathcal{C}_{L_2}(P)$ implies $\mathcal{R}_{L_2}(P) \subseteq \mathcal{R}_{L_1}(P)$. The inclusion $\mathcal{R}_{L_2}(P) \hookrightarrow \mathcal{R}_{L_1}(P)$ is functorial (it preserves determination morphisms) and is generally a proper subcategory inclusion. □

14 Conclusion

We have introduced the **Minimal Runtime Axiom**: the principle that a decision reaches runtime if and only if no static witness can be constructed for it. We developed the formal apparatus of decision spaces, static witnesses $w_d = (\tau, \pi, \sigma)$, and the three corollaries:

1. **Runtime Irreducibility**: The irreducible runtime core consists exactly of ω-dependent decisions—those depending on user input, I/O, time, and nondeterminism.

2. **The Duality**: The size of the runtime set is proportional to the epistemic deficit of the type system, providing a quantitative metric for language comparison.

3. **The Yoneda Correspondence**: Runtime type inspection is isomorphic to failure of the Yoneda embedding, connecting the axiom to the deepest structures of category theory.

We showed that the axiom unifies a broad range of existing work—partial evaluation, multi-stage programming, dependent types, linear types, effect systems—under a single design principle. We applied it to the design of JAPL, showing how each language feature corresponds to a specific decision transfer, and extended it to AI agent architecture where design-time type checking replaces runtime capability checking.

We characterized the impossibility boundary: the ω-boundary imposed by genuinely unknowable runtime data, and the SRIP boundary imposed by Gödelian incompleteness. In practice, the ω-boundary dominates, and the goal of language design is to push all non-ω decisions to compile time.

The axiom provides language designers with a precise and generative principle:

> *Every non-ω decision at runtime is a theorem waiting to be proven.*

The history of programming languages is the history of proving more and more of these theorems. The Minimal Runtime Axiom tells us where this history is heading and how to measure our progress along the way.

References

[1] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[2] H. B. Curry and R. Feys, *Combinatory Logic*, vol. I. North-Holland, 1958.

[3] W. A. Howard, "The formulae-as-types notion of construction," in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490, Academic Press, 1980.

[4] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[5] Y. Futamura, "Partial evaluation of computation process—an approach to a compiler-compiler," *Systems, Computers, Controls*, vol. 2, no. 5, pp. 45–50, 1971.

[6] O. Kiselyov, "The design and implementation of BER MetaOCaml," in *Functional and Logic Programming* (FLOPS), Springer, 2014.

[7] R. Davies and F. Pfenning, "A modal analysis of staged computation," *Journal of the ACM*, vol. 48, no. 3, pp. 555–604, 2001.

[8] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, no. 5, pp. 552–593, 2013.

[9] U. Norell, "Dependently typed programming in Agda," in *Advanced Functional Programming* (AFP), Springer, 2009.

[10] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean theorem prover (system description)," in *CADE-25*, Springer, 2015.

[11] N. D. Matsakis and F. S. Klock II, "The Rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[12] P. Wadler, "Theorems for free!," in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (FPCA), pp. 347–359, ACM, 1989.

[13] J. C. Reynolds, "Types, abstraction and parametric polymorphism," in *Information Processing 83*, pp. 513–523, 1983.

[14] J.-Y. Girard, "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur," Ph.D. thesis, Université Paris VII, 1972.

[15] P. Martin-Löf, *Intuitionistic Type Theory*. Bibliopolis, 1984.

[16] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, vol. 76, no. 2-3, pp. 95–120, 1988.

[17] G. D. Plotkin, "LCF considered as a programming language," *Theoretical Computer Science*, vol. 5, no. 3, pp. 223–255, 1977.

[18] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *Scheme and Functional Programming Workshop*, 2006.

[19] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen, "Is sound gradual typing dead?," in *POPL*, pp. 456–468, ACM, 2016.

[20] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[21] D. Walker, "Substructural type systems," in *Advanced Topics in Types and Programming Languages*, ch. 1, MIT Press, 2005.

[22] K. Honda, "Types for dyadic interaction," in *CONCUR*, Springer, 1993.

[23] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *ESOP*, Springer, 1998.

[24] A. Bauer and M. Pretnar, "Programming with algebraic effects and handlers," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 108–123, 2015.

[25] G. Plotkin and M. Pretnar, "Handlers of algebraic effects," in *ESOP*, Springer, 2009.

[26] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[27] K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik und Physik*, vol. 38, pp. 173–198, 1931.

[28] S. Mac Lane, *Categories for the Working Mathematician*. Springer, 1971.

[29] S. Awodey, *Category Theory*, 2nd ed. Oxford University Press, 2010.

[30] E. Riehl, *Category Theory in Context*. Dover, 2017.

[31] N. Yoneda, "On the homology theory of modules," *Journal of the Faculty of Science, University of Tokyo*, vol. 7, pp. 193–227, 1954.

[32] M. Long, "The Self-Reference Incompleteness Principle: Fundamental limits of formal self-knowledge," *YonedaAI Research*, 2025.

[33] W. W. Tait, "Intensional interpretations of functionals of finite type I," *Journal of Symbolic Logic*, vol. 32, no. 2, pp. 198–212, 1967.

[34] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge University Press, 1989.