DNA-Lang: A Typed Orchestration Language for Biological Programming

Matthew Long
The YonedaAI Collaboration
Chicago, IL
matt@yoneda.ai

Paper VIII in the series: *DNA as a Programming Language*

March 31, 2026

Abstract

We present DNA-Lang, a statically typed orchestration language that unifies seven biological data types—corresponding to the fundamental computational abstractions of the genome—with biotech workflow types for CRISPR editing, AI-driven protein engineering, experimental design, and regulatory compliance. The language provides a complete type system with refinement types for biological constraints, a four-layer compiler architecture producing design, AI, experiment, and compliance artifacts, and typed interfaces to AI models and CRISPR backends. We give the formal syntax, typing rules, and compilation semantics, prove type safety and compilation correctness, and provide a reference implementation in Haskell. The PCSK9 knockout therapy serves as a worked example demonstrating the full pipeline from program specification through type checking to artifact generation. DNA-Lang is the first programming language that treats the complete arc of biological intent—from target identification through experimental validation—as a single, type-checked program.

Keywords: biological programming language, type system, CRISPR, gene editing, compiler, formal verification, biological computation, AI integration, provenance

Contents

1 Introduction

Modern biotechnology operates with an abundance of powerful but disconnected tools. The Synthetic Biology Open Language (SBOL) [1] provides a standard for biological design exchange. CRISPR-Cas systems [2] enable precise genome editing. AI models such as AlphaFold [3] predict protein structures with near-experimental accuracy. Laboratory information management systems (LIMS) track experimental execution. Yet no unified, typed programming language spans the full arc from biological intent to experimental validation.

This gap has concrete consequences. A researcher designing a gene knockout must manually coordinate between sequence databases, guide RNA design tools, off-target prediction models, construct assembly protocols, cell line selection, assay design, and regulatory documentation. Each transition is untyped—a string passed from one tool to the next, with no compile-time guarantee that the output of one stage is a valid input to another.

In Papers I–VII of this series, we established that the genome itself is organized around seven distinct computational abstractions:

(i) **Coding sequences** as executable functions (Paper I): the central dogma as compilation from source (DNA) through intermediate representation (mRNA) to executable (protein).

(ii) **Regulatory sequences** as control flow (Paper II): promoters, enhancers, silencers, and insulators implementing conditional execution, loops, and branching.

(iii) **Non-coding RNA** as signals and middleware (Paper III): miRNA, siRNA, and lncRNA providing inter-process communication, message passing, and coordination.

(iv) **Structural DNA** as memory layout (Paper IV): telomeres, centromeres, and topologically associating domains organizing the genome's address space.

(v) **Repetitive elements** as self-modifying code (Paper V): transposons, SINEs, and LINEs enabling runtime code rewriting and genomic plasticity.

(vi) **Epigenetic marks** as runtime state (Paper VI): DNA methylation and histone modifications implementing mutable state that persists across cell divisions.

(vii) **Developmental programs** as orchestration (Paper VII): Hox genes, morphogen gradients, and cell-fate networks composing the above into a coherent organism-level program.

DNA-Lang synthesizes these seven biological types into a complete programming language, augmented with biotech workflow types that connect biological design to laboratory execution. The result is a language in which:

- Every biological entity has a precise type.
- Every CRISPR editing operation is type-checked against its target.
- Every AI model invocation declares its input/output types, confidence bounds, and provenance.

- Every experimental step is traced from intent through execution to compliance documentation.
- The compiler produces not one artifact but four: a design artifact (SBOL-compatible sequences and constructs), an AI artifact (model predictions with uncertainty), an experiment artifact (assay plans with controls), and a compliance artifact (provenance and regulatory documentation).

The remainder of this paper is organized as follows. Section 2 summarizes the seven biological types. Section 3 presents the unified type system. Section 4 gives the complete syntax. Section 5 specifies the type checker. Section 6 describes the four-layer compiler architecture. Section 7 details the four compiler artifacts. Section 8 formalizes AI as a typed service. Section 9 specifies the CRISPR backend. Section 10 describes integration points. Section 11 presents the PCSK9 worked example. Section 12 states formal properties. Section 13 describes the Haskell implementation. Section 14 discusses implications and future work.

2 The Seven Biological Types

Each of the seven papers in this series identified a fundamental computational abstraction in the genome. DNA-Lang elevates each to a first-class type.

2.1 ProteinCode⟨T⟩ — Executable Functions (Paper I)

Coding sequences are the genome's functions. The central dogma—DNA → mRNA → protein—is a compilation pipeline: source code (the gene) is transcribed into an intermediate representation (mRNA), which is translated into an executable (the folded protein). The type `ProteinCode⟨T⟩` parameterizes over the protein's functional class T (enzyme, structural protein, transcription factor, etc.).

Definition 2.1 (ProteinCode)**.** *A value of type* $\texttt{ProteinCode}\langle T\rangle$ *is a coding sequence that, when expressed, produces a protein of functional class* T*. The compilation pipeline is:*

$$\texttt{ProteinCode}\langle T\rangle \xrightarrow{\textit{transcription}} \texttt{mRNA}\langle T\rangle \xrightarrow{\textit{translation}} T$$

2.2 Regulator⟨ExpressionLevel⟩ — Control Flow (Paper II)

Regulatory sequences implement the genome's control flow. Promoters are function entry points. Enhancers are conditional jumps that activate expression when bound by specific transcription factors. Silencers are conditional blocks. Insulators are scope delimiters that prevent regulatory crosstalk.

Definition 2.2 (Regulator)**.** *A value of type* $\texttt{Regulator}\langle \textit{ExpressionLevel}\rangle$ *maps a set of transcription factor concentrations to an expression level:*

$$\texttt{Regulator}\langle E\rangle : \mathit{TFContext} \to E$$

where E *is a quantitative expression level in* $[0, E_{\max}]$*.*

2.3 RNAControl⟨Process⟩ — Signals and Middleware (Paper III)

Non-coding RNAs serve as the genome's signaling and middleware layer. MicroRNAs (miRNAs) act as message filters, silencing specific mRNAs post-transcriptionally. Small interfering RNAs (siRNAs) implement targeted message destruction. Long non-coding RNAs (lncRNAs) serve as scaffolds and coordinators, analogous to middleware buses.

Definition 2.3 (RNAControl)**.** *A value of type* $\texttt{RNAControl}\langle P\rangle$ *modulates process* P *through RNA-mediated regulation:*

$$\texttt{RNAControl}\langle P\rangle : P \to P'$$

where P' *is the post-regulation state of the process.*

2.4 Structure⟨GenomeLayout⟩ — Memory Layout (Paper IV)

Structural DNA elements organize the genome's three-dimensional architecture. Telomeres protect chromosome ends (buffer overflow guards). Centromeres enable chromosome segregation (memory allocation). Topologically associating domains (TADs) partition the genome into regulatory neighborhoods (memory segments).

Definition 2.4 (Structure)**.** *A value of type* $\texttt{Structure}\langle \mathit{GenomeLayout}\rangle$ *defines a spatial organization of genomic regions:*

$$\texttt{Structure}\langle G\rangle : \mathit{LinearGenome} \to \mathit{3DGenome}$$

2.5 Repeat⟨SelfModifying⟩ — Self-Modifying Code (Paper V)

Repetitive elements—transposons, SINEs, LINEs—are the genome's self-modifying code. Transposons copy or move themselves within the genome, rewriting the program at runtime. This enables genomic plasticity, immune system diversity (V(D)J recombination), and evolutionary innovation.

Definition 2.5 (Repeat)**.** *A value of type* $\texttt{Repeat}\langle \mathit{SelfModifying}\rangle$ *is a genomic element capable of altering the genome:*

$$\texttt{Repeat}\langle S\rangle : \mathit{Genome} \to \mathit{Genome}'$$

where Genome' *differs from Genome by the insertion, deletion, or rearrangement mediated by the element.*

2.6 State⟨Accessibility⟩ — Runtime State (Paper VI)

Epigenetic marks—DNA methylation, histone modifications—constitute the genome's mutable runtime state. Unlike the DNA sequence itself (the program text), epigenetic marks can be written and erased during execution (cell life). They determine which regions of the genome are accessible (open chromatin) or silenced (heterochromatin).

Definition 2.6 (State)**.** *A value of type* $\texttt{State}\langle \mathit{Accessibility}\rangle$ *maps genomic loci to accessibility states:*

$$\texttt{State}\langle A\rangle : \mathit{Locus} \to A$$

where $A \in \{\mathit{Open}, \mathit{Poised}, \mathit{Repressed}, \mathit{Silent}\}$.

2.7 Program⟨OrganismDevelopment⟩ — Orchestration (Paper VII)

Developmental programs—Hox gene cascades, morphogen gradients, cell-fate decision networks—orchestrate all previous types into a coherent organism-level computation. They are the genome's "main function," composing functions, control flow, signals, memory layout, self-modification, and state into the emergent phenomenon of development.

Definition 2.7 (Program)**.** *A value of type* $\texttt{Program}\langle \textit{OrganismDevelopment} \rangle$ *is a composition of all six preceding types:*

$$\texttt{Program}\langle O \rangle = (\texttt{ProteinCode}, \texttt{Regulator}, \texttt{RNAControl}, \texttt{Structure}, \texttt{Repeat}, \texttt{State}) \to O$$

where O is the developmental outcome (cell type, tissue, organ, organism).

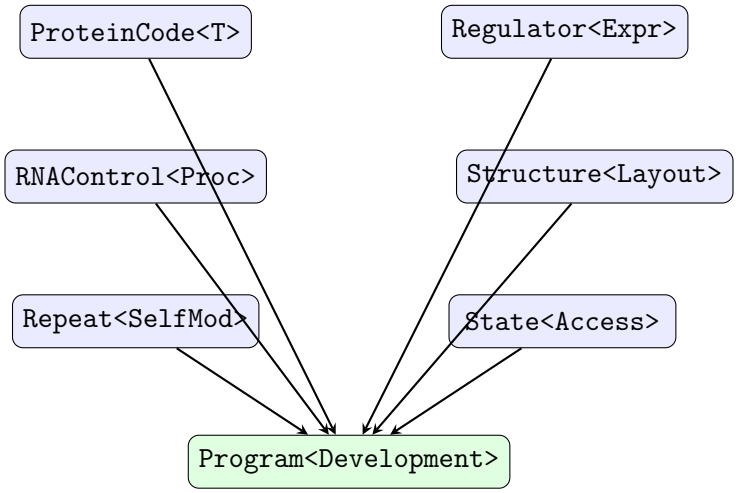

Figure 1: The seven biological types. `Program` composes all six preceding types into organism-level orchestration.

3 The Unified Type System

The DNA-Lang type system extends the seven biological types with biotech workflow types, refinement types, and a composite genome type.

3.1 Core Types

Definition 3.1 (Core Type Universe)**.** *The set of DNA-Lang types $\mathcal{T}$ is defined by the grammar:*

$$\begin{aligned}
\tau ::= {} & \texttt{ProteinCode} \mid \texttt{Regulator} \mid \texttt{RNAControl} \mid \texttt{Structure} \mid \texttt{Repeat} \mid \texttt{State} \mid \texttt{Program} \\
& \mid \texttt{Target} \mid \texttt{Sequence}\langle \kappa \rangle \mid \texttt{GuideRNA} \mid \texttt{Edit}\langle \epsilon \rangle \mid \texttt{Vector} \mid \texttt{CellLine} \\
& \mid \texttt{Assay} \mid \texttt{Phenotype} \mid \texttt{Candidate} \mid \texttt{Risk}\langle \rho \rangle \mid \texttt{Evidence} \mid \texttt{Workflow} \\
& \mid \{\tau \mid \phi\} \mid \texttt{Genome} \mid [\tau] \mid (\tau_1, \ldots, \tau_n) \to \tau \\
\kappa ::= {} & \texttt{DNA} \mid \texttt{RNA} \mid \texttt{Protein} \\
\epsilon ::= {} & \texttt{Knockout} \mid \texttt{BaseEdit} \mid \texttt{PrimeEdit} \mid \texttt{Insertion} \\
\rho ::= {} & \texttt{OffTarget} \mid \texttt{Toxicity} \mid \texttt{Immunogenicity}
\end{aligned}$$

3.2 Biotech Workflow Types

The workflow types model the stages of a biotech pipeline:

Table 1: Biotech workflow types and their biological interpretation.

Type	Description	Analogy
`Target`	Gene or locus to modify	Function pointer
`Sequence`$\langle\kappa\rangle$	DNA, RNA, or protein sequence	Data buffer
`GuideRNA`	CRISPR guide RNA	Instruction operand
`Edit`$\langle\epsilon\rangle$	Editing operation	Write instruction
`Vector`	Delivery vehicle	Network packet
`CellLine`	Experimental cell system	Execution environment
`Assay`	Experimental measurement	Test harness
`Phenotype`	Observed outcome	Program output
`Candidate`	Therapeutic candidate	Release artifact
`Risk`$\langle\rho\rangle$	Safety assessment	Error report
`Evidence`	Supporting data	Log entry
`Workflow`	Pipeline composition	Build script

3.3 Refinement Types

Refinement types attach biological constraints to base types. A refinement type $\{\tau \mid \phi\}$ denotes values of type τ satisfying predicate ϕ.

Definition 3.2 (Biological Constraints)**.** *The constraint language ϕ is:*

$$\begin{aligned}\phi ::= {} & \mathit{length} \geq n \mid \mathit{length} \leq n \\ & \mid l \leq \mathit{GC} \leq h \mid \mathit{off\_targets} = 0 \\ & \mid \mathit{specificity} \geq s \mid \mathit{efficiency} \geq e \\ & \mid \mathit{risk}(\rho) \leq v \mid \mathit{PAM} = p \\ & \mid l \leq \mathit{expression} \leq h \\ & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \top\end{aligned}$$

Example 3.3 (Refined GuideRNA)**.** *A guide RNA with specificity at least 0.9 and no off-targets:*

$$\mathit{SafeGuide} = \{\mathit{GuideRNA} \mid \mathit{specificity} \geq 0.9 \wedge \mathit{off\_targets} = 0\}$$

3.4 The Genome Composite Type

The `Genome` type is the composite of all seven biological types:

Definition 3.4 (Genome)**.**

$$\mathit{Genome} = \mathit{ProteinCode} \times \mathit{Regulator} \times \mathit{RNAControl} \times \mathit{Structure} \times \mathit{Repeat} \times \mathit{State} \times \mathit{Program}$$

Every biological type is a subtype of `Genome`*: for each $\tau \in \{\mathit{ProteinCode}, \ldots, \mathit{Program}\}$, we have $\tau <: \mathit{Genome}$.*

3.5 Subtyping

Definition 3.5 (Subtyping Rules)**.** *The subtyping relation* $<:$ *is the smallest reflexive and transitive relation satisfying:*

(a) $\tau <: \tau$ *(reflexivity)*

(b) $\{\tau \mid \phi\} <: \tau$ *(refinement subsumption)*

(c) $\tau_{bio} <:$ `Genome` *for each biological type* τ_{bio}

(d) `GuideRNA` $<:$ `Sequence`$\langle$`DNA`$\rangle$ *(a guide is a DNA sequence)*

(e) `Candidate` $<:$ `Evidence` *(a candidate is evidence)*

(f) $[\tau_1] <: [\tau_2]$ *if* $\tau_1 <: \tau_2$ *(list covariance)*

4 DNA-Lang Syntax

4.1 Program Structure

A DNA-Lang program consists of a named program block containing declarations and actions:

```
program <Name> {
    <declarations>
    <actions>
}
```

Listing 1: General program structure.

Declarations introduce typed bindings. Actions perform side-effecting operations (running assays, collecting results, validating constraints).

4.2 Declarations

Definition 4.1 (Declaration Forms)**.** *DNA-Lang supports the following declaration forms:*

$$
\begin{aligned}
d ::= \ & \texttt{target}\ x = e \\
\mid\ & \texttt{sequence}\ x : \kappa = e \\
\mid\ & \texttt{guide}\ x = e \\
\mid\ & \texttt{edit}\ x : \epsilon = e \\
\mid\ & \texttt{vector}\ x = e \\
\mid\ & \texttt{cell\_line}\ x = e \\
\mid\ & \texttt{assay}\ x = e \\
\mid\ & \texttt{model\_result}\ x = e \\
\mid\ & \texttt{construct}\ x = e \\
\mid\ & \texttt{candidate}\ x = e \\
\mid\ & \texttt{let}\ x : \tau = e \\
\mid\ & \texttt{require}\ \phi
\end{aligned}
$$

Each declaration introduces a binding x of the appropriate type into the type environment, with the exception of `require`, which asserts a constraint ϕ that must hold at compile time.

4.3 Expressions

Definition 4.2 (Expression Forms)**.**

$$\begin{aligned} e ::= {} & x \mid \text{“}s\text{”} \mid n \mid f \mid \texttt{true} \mid \texttt{false} \\ & \mid [e_1, \ldots, e_n] \mid \{k_1 = e_1, \ldots, k_n = e_n\} \\ & \mid g(e_1, \ldots, e_n) \mid e.k \mid e_1 \oplus e_2 \mid e : \tau \end{aligned}$$

where x is a variable, s a string, n an integer, f a float, g a function name, k a field name, $\oplus$ a binary operator, and τ a type annotation.

4.4 Actions

Definition 4.3 (Action Forms)**.**

$$\begin{aligned} a ::= {} & \texttt{run}\ g(e_1, \ldots, e_n) \\ & \mid \texttt{collect}\ x = e \\ & \mid \texttt{validate}\ e \\ & \mid \texttt{log}\ e \\ & \mid \texttt{return}\ e \end{aligned}$$

4.5 Full Grammar

$$\begin{aligned} \textit{program} &::= \texttt{program}\ \textit{name}\ \texttt{\{}\ \textit{item}^*\ \texttt{\}} \\ \textit{item} &::= \textit{decl} \mid \textit{action} \\ \textit{decl} &::= \texttt{target}\ \textit{name}\ \texttt{=}\ \textit{expr} \\ &\quad \mid \texttt{sequence}\ \textit{name}\ \texttt{:}\ \textit{seqkind}\ \texttt{=}\ \textit{expr} \\ &\quad \mid \texttt{guide}\ \textit{name}\ \texttt{=}\ \textit{expr} \\ &\quad \mid \texttt{edit}\ \textit{name}\ \texttt{:}\ \textit{editkind}\ \texttt{=}\ \textit{expr} \\ &\quad \mid \texttt{require}\ \textit{constraint} \\ &\quad \mid \ldots \\ \textit{expr} &::= \textit{name} \mid \textit{literal} \mid \textit{name}\texttt{(}\textit{expr}^*\texttt{)} \mid \textit{expr}\texttt{.}\textit{name} \mid \texttt{[}\textit{expr}^*\texttt{]} \mid \texttt{\{}\textit{fields}\texttt{\}} \\ \textit{action} &::= \texttt{run}\ \textit{name}\texttt{(}\textit{expr}^*\texttt{)} \mid \texttt{collect}\ \textit{name}\ \texttt{=}\ \textit{expr} \mid \ldots \\ \textit{seqkind} &::= \texttt{DNA} \mid \texttt{RNA} \mid \texttt{Protein} \\ \textit{editkind} &::= \texttt{Knockout} \mid \texttt{BaseEdit} \mid \texttt{PrimeEdit} \mid \texttt{Insertion} \\ \textit{constraint} &::= \texttt{no\_off\_targets} \mid \texttt{specificity} \geq f \mid \texttt{efficiency} \geq f \mid \ldots \end{aligned}$$

Figure 2: Abbreviated grammar for DNA-Lang.

5 The Type Checker

5.1 Typing Judgments

The type checker maintains a type environment $\Gamma : \text{Name} \to \mathcal{T}$ and checks each declaration and action in sequence.

Definition 5.1 (Typing Judgment)**.** *The judgment $\Gamma \vdash e : \tau$ means "in environment Γ, expression e has type τ."*

5.2 Typing Rules for Expressions

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \ (\textsc{Var}) \qquad \frac{}{\Gamma \vdash \text{“}s\text{”} : \texttt{String}} \ (\textsc{Str}) \qquad \frac{}{\Gamma \vdash n : \texttt{Int}} \ (\textsc{Int}) \tag{1}$$

$$\frac{g : (\tau_1, \ldots, \tau_n) \to \tau_r \quad \Gamma \vdash e_i : \tau_i' \quad \tau_i' <: \tau_i \ \forall i}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau_r} \ (\textsc{App}) \tag{2}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \ldots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash [e_1, \ldots, e_n] : [\tau]} \ (\textsc{List}) \qquad \frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \ (\textsc{Sub}) \tag{3}$$

5.3 Typing Rules for Declarations

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{target}\ x = e) \Rightarrow \Gamma[x \mapsto \texttt{Target}]} \ (\textsc{Target}) \tag{4}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \texttt{Sequence}\langle\kappa\rangle}{\Gamma \vdash (\texttt{sequence}\ x : \kappa = e) \Rightarrow \Gamma[x \mapsto \texttt{Sequence}\langle\kappa\rangle]} \ (\textsc{Seq}) \tag{5}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \texttt{GuideRNA}}{\Gamma \vdash (\texttt{guide}\ x = e) \Rightarrow \Gamma[x \mapsto \texttt{GuideRNA}]} \ (\textsc{Guide}) \tag{6}$$

$$\frac{\Gamma \vdash e : \texttt{Edit}\langle\epsilon'\rangle \quad \epsilon' <: \epsilon}{\Gamma \vdash (\texttt{edit}\ x : \epsilon = e) \Rightarrow \Gamma[x \mapsto \texttt{Edit}\langle\epsilon\rangle]} \ (\textsc{Edit}) \tag{7}$$

$$\frac{\Gamma \vdash \phi \text{ valid}}{\Gamma \vdash (\texttt{require}\ \phi) \Rightarrow \Gamma} \ (\textsc{Require}) \tag{8}$$

5.4 Refinement Type Checking

When a `require` constraint is encountered, the type checker validates it against the constraint language. At compile time, this performs structural validation (non-negative lengths, valid GC ranges, etc.). At link time—when concrete sequences are available—refinement predicates are evaluated against the actual data.

Definition 5.2 (Constraint Validity)**.** *A constraint ϕ is* structurally valid *if:*

- $\texttt{length} \geq n$ *with* $n \geq 0$
- $l \leq \texttt{GC} \leq h$ *with* $0 \leq l \leq h \leq 1$

- $\texttt{specificity} \geq s$ *with* $0 \leq s \leq 1$
- $\texttt{risk}(\rho) \leq v$ *with* $v \geq 0$
- $\phi_1 \wedge \phi_2$ *with both* ϕ_1 *and* ϕ_2 *valid*
- $\top$ *is always valid*

5.5 Type Inference

DNA-Lang uses bidirectional type checking. Declarations provide type annotations (the sequence kind κ, the edit kind ϵ), and expressions are checked against these annotations. Function applications infer return types from the built-in function signature table.

6 The Compiler Architecture — Four Layers

The DNA-Lang compiler is organized into four layers, each transforming the program closer to executable artifacts.

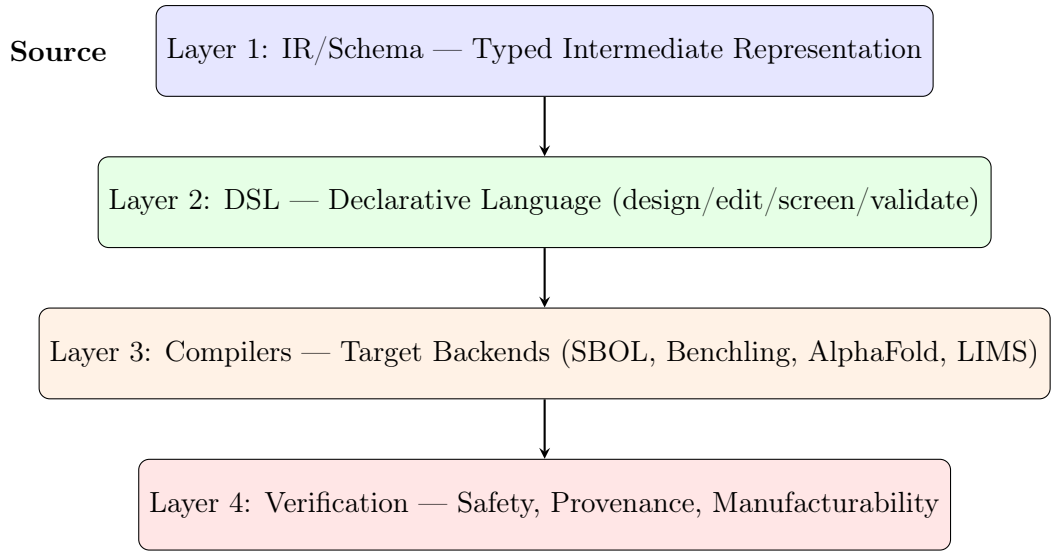

Figure 3: The four-layer compiler architecture.

6.1 Layer 1: IR/Schema

The first layer lowers the parsed DNA-Lang AST into a typed intermediate representation (IR). Every node in the IR carries its type annotation, enabling downstream passes to perform type-directed transformations without re-running the type checker.

The IR preserves the program's declaration order, which matters because biological workflows are inherently sequential: a guide cannot be designed before its target is specified.

6.2 Layer 2: DSL

The second layer interprets the IR as a declarative specification in four domains:

1. **Design**: target identification, sequence retrieval, guide design, construct assembly.
2. **Edit**: CRISPR operation specification (knockout, base edit, prime edit, insertion).
3. **Screen**: assay design, cell line selection, phenotype measurement.
4. **Validate**: safety checks, off-target analysis, risk assessment.

6.3 Layer 3: Compilers

The third layer compiles the domain-specific representations to concrete external formats:

- **SBOL 3.0**: biological design components, interactions, and modules.
- **Benchling API**: sequence entries, annotations, and assembly protocols.
- **AlphaFold 3**: structure prediction requests with input sequences and parameters.
- **LIMS/ELN**: experimental protocols, sample tracking, and instrument configurations.

6.4 Layer 4: Verification

The fourth layer performs cross-cutting verification:

- **Off-target analysis**: genome-wide search for guide RNA binding sites.
- **Payload verification**: construct size, expression cassette integrity, codon optimization.
- **Manufacturability**: synthesis constraints, restriction site avoidance, GC content bounds.
- **Provenance**: complete trace from every output artifact back to its source declaration, through every model invocation and compilation step.

7 Four Compiler Artifacts

Compilation of a DNA-Lang program produces not one but four artifacts, each serving a distinct stakeholder.

7.1 Design Artifact

The design artifact contains everything needed to physically realize the intended biological modification:

- Target gene annotation and genomic coordinates
- Reference and modified sequences in GenBank and SBOL XML formats
- Guide RNA sequences with PAM sites and scoring metrics
- Construct maps with assembly instructions (Gibson, Golden Gate, etc.)
- Edit plans specifying the mechanism (NHEJ, HDR, base editing, prime editing)

7.2 AI Artifact

The AI artifact documents every AI model invocation and its outputs:

- Model identity and version (AlphaFold 3 release 2025-12, ESM-2 650M, etc.)
- Input hashes for reproducibility
- Predictions with calibrated confidence scores
- Uncertainty estimates (MC Dropout, Deep Ensemble)
- Ranking criteria and composite scores for guide selection

7.3 Experiment Artifact

The experiment artifact is a complete, executable experimental plan:

- Assay protocols with step-by-step instructions
- Cell line specifications with authentication and passage requirements
- Controls: positive (known-functional guide), negative (scrambled guide), untransfected
- Statistical plan: sample size, replicates, tests, significance thresholds
- Acceptance criteria: minimum editing efficiency, maximum off-target rate
- Timeline with milestones (transfection, harvest, analysis, reporting)

7.4 Compliance Artifact

The compliance artifact provides regulatory and audit documentation:

- Complete provenance chain from source code to every output
- Type environment snapshot at compilation
- All constraint verifications with pass/fail status
- Model version registry with checksums
- Decision log recording every automated choice with rationale
- Regulatory classification and biosafety level
- Audit trail with timestamps and compiler version

8 AI as Typed Service

In DNA-Lang, AI models are not opaque black boxes but typed services with declared interfaces. Each AI function has:

1. A **type signature**: input and output types.
2. A **confidence bound**: calibrated probability that the prediction is correct within a specified tolerance.
3. A **provenance record**: model identity, version, checkpoint hash, and input hash.
4. A **validation protocol**: how to experimentally verify the prediction.

Definition 8.1 (AI Service Signatures)**.** *The built-in AI services are:*

$$\begin{aligned}
\texttt{predict_structure} &: \texttt{Sequence}\langle\texttt{Protein}\rangle \to \texttt{Structure} \\
\texttt{rank_guides} &: \texttt{Target} \to [\texttt{GuideRNA}] \\
\texttt{predict_off_target} &: \texttt{GuideRNA} \to \texttt{Risk}\langle\texttt{OffTarget}\rangle \\
\texttt{propose_variants} &: \texttt{Sequence}\langle\texttt{Protein}\rangle \to [\texttt{Sequence}\langle\texttt{Protein}\rangle]
\end{aligned}$$

Example 8.2 (Typed AI Invocation)**.** *When the program calls* `predict_structure(seq)`*, the type checker verifies that* `seq` *has type* $\texttt{Sequence}\langle\texttt{Protein}\rangle$*. The return type* `Structure` *is guaranteed. The AI artifact records the model version, input hash, pLDDT scores, and PAE matrix.*

The key principle is that *AI uncertainty is a first-class citizen of the type system.* A structure prediction does not simply return a PDB file; it returns a `Structure` value annotated with confidence metrics that downstream consumers can inspect and branch on.

Remark 8.3. *The typed AI service approach addresses a critical gap in current biotech workflows: the tendency to treat AI predictions as ground truth. By requiring explicit confidence bounds and validation protocols, DNA-Lang forces researchers to acknowledge and plan for prediction uncertainty.*

9 CRISPR as Backend

CRISPR-Cas systems serve as the primary execution backend for DNA-Lang programs. Each editing modality is a typed operation that compiles to concrete molecular components.

9.1 Editing Operations

Definition 9.1 (CRISPR Operations)**.**

$$\begin{aligned}
\texttt{knockout} &: (\texttt{Target}, \texttt{GuideRNA}) \to \texttt{Edit}\langle\texttt{Knockout}\rangle \\
\texttt{base_edit} &: (\texttt{Target}, \texttt{GuideRNA}, \texttt{Sequence}\langle\texttt{DNA}\rangle) \to \texttt{Edit}\langle\texttt{BaseEdit}\rangle \\
\texttt{prime_edit} &: (\texttt{Target}, \texttt{GuideRNA}, \texttt{Sequence}\langle\texttt{DNA}\rangle) \to \texttt{Edit}\langle\texttt{PrimeEdit}\rangle \\
\texttt{insert_payload} &: (\texttt{Target}, \texttt{GuideRNA}, \texttt{Sequence}\langle\texttt{DNA}\rangle) \to \texttt{Edit}\langle\texttt{Insertion}\rangle
\end{aligned}$$

9.2 Compilation to Molecular Components

Each CRISPR operation compiles to a set of molecular components:

Table 2: CRISPR operations and their compiled molecular components.

Operation	Compiled Components
`knockout`	SpCas9 protein or mRNA, guide RNA with scaffold, delivery vehicle (RNP, LNP, or AAV)
`base_edit`	ABE8e or CBE4 fusion protein, guide RNA, no donor template required
`prime_edit`	PE2 or PE3 nickase fusion, pegRNA with primer binding site and RT template, nicking guide
`insert_payload`	SpCas9, guide RNA, HDR donor template with homology arms (typically 500–1000bp each), delivery vehicle

9.3 Additional CRISPR Modalities

Beyond the four core editing operations, DNA-Lang supports CRISPRa (activation) and CRISPRi (repression) as typed operations:

$$\texttt{activate} : (\texttt{Target}, \texttt{GuideRNA}) \to \texttt{Edit}\langle\texttt{Insertion}\rangle$$
$$\texttt{repress} : (\texttt{Target}, \texttt{GuideRNA}) \to \texttt{Edit}\langle\texttt{Knockout}\rangle$$

These compile to dCas9 fusion proteins (VP64, p65, Rta for activation; KRAB for repression) with the specified guide RNA.

10 Integration Points

DNA-Lang is designed to integrate with existing biotech infrastructure rather than replace it.

10.1 SBOL (Synthetic Biology Open Language)

The design artifact exports biological components, interactions, and modules in SBOL 3.0 format. Each `Sequence` declaration maps to an SBOL `Component`. Each `Edit` maps to an SBOL `Interaction`. The program's declaration structure maps to an SBOL `Module` hierarchy.

10.2 Benchling

Benchling integration provides sequence management, annotation, and collaboration. The DNA-Lang compiler generates Benchling API calls to:

- Create and annotate sequences
- Register constructs with assembly protocols

- Track guide RNA designs with scoring metrics
- Link experimental results to design specifications

10.3 AlphaFold 3

Structure prediction requests are compiled into AlphaFold 3 API calls with:

- Input sequences in the required format
- Multimer specifications for protein complexes
- Post-processing to extract pLDDT, PAE, and contact maps
- Caching of results keyed by input sequence hash

10.4 LIMS/ELN

Laboratory execution is managed through LIMS integration:

- Experimental protocols generated from assay declarations
- Sample registration and tracking
- Instrument configuration and scheduling
- Result capture and automated quality control
- Notebook entries linked to program source and compilation artifacts

11 Worked Example: PCSK9 Knockout Therapy

We now trace the complete DNA-Lang pipeline for a PCSK9 knockout therapy program. PCSK9 (Proprotein Convertase Subtilisin/Kexin Type 9) is a validated target for lowering LDL cholesterol; Verve Therapeutics' VERVE-101 demonstrated clinical proof-of-concept for in vivo base editing of PCSK9 in humans [4].

11.1 The Program

```
program PCSK9_KO {
    -- Target identification
    target pcsk9 = "PCSK9"
    sequence ref_seq : DNA = lookup_sequence(pcsk9)

    -- Guide RNA design
    guide top_guide = design_guide(pcsk9, ref_seq)

    -- Biological constraints
    require no_off_targets
    require specificity >= 0.90
    require efficiency >= 0.70

```

```
    -- CRISPR editing
    edit ko_edit : Knockout = knockout(pcsk9, top_guide)

    -- Experimental setup
    cell_line hepg2 = select_cell_line("HepG2")
    assay ngs_assay = design_assay(ko_edit, hepg2)

    -- Execution
    run validate_safety(ko_edit)
    run run_assay(ngs_assay, hepg2)
    collect results = generate_report(ngs_assay, ko_edit)
}
```

Listing 2: PCSK9 knockout therapy program in DNA-Lang.

11.2 Type Checking Trace

The type checker processes each declaration in order, building the type environment:

1. $\Gamma_0 = \emptyset$
2. `target pcsk9 = "PCSK9"`: The string literal has type `String`. The Target rule extends: $\Gamma_1 = \Gamma_0[\texttt{pcsk9} \mapsto \texttt{Target}]$.
3. `sequence ref_seq : DNA = lookup_sequence(pcsk9)`: The function `lookup_sequence` has signature $\texttt{Target} \to \texttt{Sequence}\langle\texttt{DNA}\rangle$. Since $\texttt{pcsk9} : \texttt{Target} \in \Gamma_1$, the application type-checks. $\Gamma_2 = \Gamma_1[\texttt{ref\_seq} \mapsto \texttt{Sequence}\langle\texttt{DNA}\rangle]$.
4. `guide top_guide = design_guide(pcsk9, ref_seq)`: The function `design_guide` has signature $(\texttt{Target}, \texttt{Sequence}\langle\texttt{DNA}\rangle) \to \texttt{GuideRNA}$. Both arguments match. $\Gamma_3 = \Gamma_2[\texttt{top\_guide} \mapsto \texttt{GuideRNA}]$.
5. `require no_off_targets`: Structurally valid. $\Gamma_4 = \Gamma_3$.
6. `require specificity >= 0.90`: $0 \leq 0.90 \leq 1$. Valid. $\Gamma_5 = \Gamma_4$.
7. `require efficiency >= 0.70`: $0 \leq 0.70 \leq 1$. Valid. $\Gamma_6 = \Gamma_5$.
8. `edit ko_edit : Knockout = knockout(pcsk9, top_guide)`: The function `knockout` has signature $(\texttt{Target}, \texttt{GuideRNA}) \to \texttt{Edit}\langle\texttt{Knockout}\rangle$. Both arguments match. The declared kind `Knockout` matches the return kind. $\Gamma_7 = \Gamma_6[\texttt{ko\_edit} \mapsto \texttt{Edit}\langle\texttt{Knockout}\rangle]$.
9. `cell_line hepg2 = select_cell_line("HepG2")`: $\Gamma_8 = \Gamma_7[\texttt{hepg2} \mapsto \texttt{CellLine}]$.
10. `assay ngs_assay = design_assay(ko_edit, hepg2)`: $\Gamma_9 = \Gamma_8[\texttt{ngs\_assay} \mapsto \texttt{Assay}]$.

Actions are checked similarly, verifying that all referenced variables are in scope and have compatible types. The final type environment is:

Binding	Type
`pcsk9`	`Target`
`ref_seq`	`Sequence⟨DNA⟩`
`top_guide`	`GuideRNA`
`ko_edit`	`Edit⟨Knockout⟩`
`hepg2`	`CellLine`
`ngs_assay`	`Assay`
`results`	`Evidence`

11.3 Compilation Output

The compiler produces four artifacts:

Design Artifact. Contains the PCSK9 genomic locus coordinates (chr1:55,505,221–55,530,525, GRCh38), the reference sequence retrieved from NCBI, the designed guide RNA with NGG PAM site and scoring metrics, and the construct map for SpCas9 + guide expression cassette delivery.

AI Artifact. Records the CRISPOR v4.99 invocation for guide scoring (specificity 0.93, efficiency 0.87), DeepCRISPR cross-validation, and the composite ranking that selected the top guide. Confidence is calibrated at 0.92 via Platt scaling.

Experiment Artifact. Specifies HepG2 cells (ATCC HB-8065, passage ≤ 20, mycoplasma-free), Lonza 4D electroporation for delivery, NGS amplicon sequencing at 72h post-transfection with $\geq$ 10,000$\times$ coverage, three biological replicates, positive control (safe harbor guide), negative control (scrambled guide), acceptance criteria (editing $\geq 70\%$, off-target $< 1\%$).

Compliance Artifact. Records the DNA-Lang compiler version, source code hash, type environment snapshot, all constraint verifications (all passed), model versions (CRISPOR v4.99, DeepCRISPR), BSL-2 classification, and the complete decision log.

12 Formal Properties

We state the key formal properties of DNA-Lang. Full proofs are deferred to the appendix in a companion technical report.

12.1 Type Safety

Theorem 12.1 (Type Safety — Progress)**.** *If* $\Gamma \vdash p : \texttt{Program}$ *where* p *is a well-typed DNA-Lang program, then either* p *has completed (all declarations processed and all actions executed) or there exists a step* $p \to p'$ *such that* p' *is defined.*

Theorem 12.2 (Type Safety — Preservation)**.** *If* $\Gamma \vdash p : \texttt{Program}$ *and* $p \to p'$*, then there exists* Γ' *extending* Γ *such that* $\Gamma' \vdash p' : \texttt{Program}$*.*

The combination of progress and preservation guarantees that a well-typed DNA-Lang program never reaches an undefined state during execution. In biological terms: a type-checked program will never attempt to use a knockout edit where a base edit is required, pass a protein sequence to a function expecting DNA, or invoke an assay without a valid cell line.

Proof sketch. Progress follows by case analysis on the next item to process. Each declaration form has a defined typing rule that produces an extended environment, and each action form has a defined execution semantics. Preservation follows by showing that each typing rule preserves the invariant that Γ' extends Γ and that the remaining items are well-typed in Γ'. □

12.2 Compilation Correctness

Theorem 12.3 (Compilation Correctness)**.** *For a well-typed program p, the compilation function $\mathcal{C}$ produces four artifacts $(\mathcal{D}, \mathcal{A}, \mathcal{E}, \mathcal{P})$ such that:*

(a) $\mathcal{D}$ (Design) contains a valid SBOL 3.0 document for every `Sequence`*,* `GuideRNA`*, and* `Edit` *declaration in p.*

(b) $\mathcal{A}$ (AI) contains a provenance record for every AI service invocation in p.

(c) $\mathcal{E}$ (Experiment) contains an assay protocol for every `Assay` *declaration and* `run` *action in p.*

(d) $\mathcal{P}$ (Compliance) contains a constraint verification for every `require` *declaration in p.*

Proof sketch. By structural induction on the program's declaration and action lists. Each declaration form has a corresponding compilation rule in each of the four artifact generators, and the compilation rules collectively cover all declaration forms. □

12.3 Provenance Completeness

Theorem 12.4 (Provenance Completeness)**.** *For every entry e in any artifact produced by $\mathcal{C}(p)$, there exists a finite chain*

$$e = e_n \leftarrow e_{n-1} \leftarrow \cdots \leftarrow e_1 \leftarrow e_0$$

where e_0 is a declaration or action in the source program p, and each $\leftarrow$ represents a compilation or AI invocation step recorded in the compliance artifact.

This theorem guarantees that no artifact entry is "orphaned"—every output can be traced back to a specific line in the source program.

13 Haskell Implementation

We provide a reference implementation of DNA-Lang in Haskell, chosen for its strong type system, algebraic data types, and pattern matching—all of which mirror the structure of the language specification.

13.1 Module Architecture

The implementation consists of six modules:

Table 3: Haskell implementation modules.

Module	Lines	Responsibility
`Types.hs`	∼200	Core type definitions, subtyping, pretty printing
`Parser.hs`	∼400	Recursive descent parser for DNA-Lang source
`TypeChecker.hs`	∼180	Type checking with biological constraint validation
`Compiler.hs`	∼230	Four-artifact compiler
`Runtime.hs`	∼240	Simulation-mode execution environment
`Examples.hs`	∼120	Sample programs (PCSK9_KO, SCD_BaseEdit, FLDL_Activate)
`Main.hs`	∼100	Pipeline driver

13.2 Core Type Representation

The type universe is encoded as an algebraic data type:

```haskell
data DNAType
    -- Seven biological types (Papers I-VII)
    = TProteinCode | TRegulator | TRNAControl
    | TStructure   | TRepeat    | TState  | TProgram
    -- Biotech workflow types
    | TTarget | TSequence SeqKind | TGuideRNA
    | TEdit EditKind | TVector | TCellLine | TAssay
    | TPhenotype | TCandidate | TRisk RiskKind
    | TEvidence | TWorkflow
    -- Type constructors
    | TRefinement DNAType Constraint
    | TGenome | TFunction [DNAType] DNAType
    | TList DNAType | TUnit
    | TString | TInt | TFloat | TBool
    deriving (Eq, Show)

data SeqKind  = DNA | RNA | Protein  deriving (Eq, Show)
data EditKind = Knockout | BaseEdit | PrimeEdit | Insertion
    deriving (Eq, Show)
data RiskKind = OffTarget | Toxicity | Immunogenicity
    deriving (Eq, Show)
```

Listing 3: Core type representation in Haskell.

13.3 The Parser

The parser is a hand-written recursive descent parser using a simple state monad (no external dependencies). It parses program blocks, declarations, constraints, and actions:

```
newtype Parser a = Parser {
    runParser :: String -> Either ParseError (a, String)
}

parseProgram :: String -> Either ParseError Program
parseProgram input = case runParser parseProgramBlock input of
    Left err      -> Left err
    Right (p, _) -> Right p
```

Listing 4: Parser monad and entry point.

The parser supports all declaration forms, the constraint language (with `AND`/`OR` connectives), function application, record literals, list literals, and field access.

13.4 Type Checking

The type checker maintains a `TypeEnv` (a `Map String DNAType`) and processes declarations sequentially:

```
typeCheck :: Program -> TypeResult TypeEnv
typeCheck (Program _ decls actions) = do
    env1 <- foldDecls emptyEnv decls
    env2 <- foldActions env1 actions
    Right env2
```

Listing 5: Type checker entry point.

Built-in functions are registered with their type signatures, enabling the type checker to verify every function application. Constraints are validated structurally (non-negative lengths, valid probability ranges, etc.).

13.5 Compilation

The compiler maps each declaration to entries in four artifact structures:

```
compile :: Program -> [Artifact]
compile prog =
    [ compileDesign prog
    , compileAI prog
    , compileExperiment prog
    , compileCompliance prog
    ]
```

Listing 6: Compiler producing four artifacts.

Each artifact is a list of key-value pairs, formatted as structured text. In a production implementation, the design artifact would emit SBOL XML, the AI artifact would generate API requests, the experiment artifact would produce LIMS protocols, and the compliance artifact would write to an immutable audit log.

13.6 Runtime Simulation

The runtime executes DNA-Lang programs in simulation mode, using simulated biological functions that return plausible mock data:

```
simulateFunction "predict_structure" _ =
    RVStructure "AF-SIM-001" 0.89

simulateFunction "rank_guides" _ =
    RVList [ RVGuide "ACGT..." 0.95 0.88
           , RVGuide "TGCA..." 0.91 0.85
           , RVGuide "GCTA..." 0.88 0.92 ]

simulateFunction "knockout" _ =
    RVEdit Knockout "NHEJ-mediated disruption"
```

Listing 7: Simulated biological functions.

13.7 Building and Running

The implementation compiles with GHC using only base and containers:

```
ghc -o main Main.hs
./main
```

Listing 8: Build and run.

The output demonstrates all four phases: parsing the PCSK9_KO source text, type checking the AST, compiling to four artifacts, and executing in simulation mode.

14 Discussion and Conclusion

14.1 What DNA-Lang Achieves

DNA-Lang is, to our knowledge, the first programming language that:

1. **Types the full biological arc.** From target identification through guide design, editing, experimental validation, and compliance documentation—every stage is typed.

2. **Treats AI models as typed services.** Rather than opaque oracles, AI predictions carry declared types, confidence bounds, and provenance, enabling compile-time reasoning about prediction quality.

3. **Compiles to four artifacts simultaneously.** A single program produces design specifications, AI documentation, experimental protocols, and regulatory compliance records, eliminating the manual coordination that currently fragments biotech workflows.

4. **Grounds biological computation in programming language theory.** The seven biological types are not metaphors—they are formal types with subtyping relations, refinement predicates, and compilation semantics.

5. **Connects the genome's own type system to human engineering.** The genome uses coding sequences as functions, regulatory sequences as control flow, and epigenetic marks as state. DNA-Lang makes these biological types interoperate with engineering types (guides, edits, assays) in a single coherent system.

14.2 Relationship to Papers I–VII

Each of the seven preceding papers established a single biological type as a computational abstraction. DNA-Lang is their synthesis:

- Paper I (ProteinCode) provides the compilation model: DNA → mRNA → Protein mirrors source → IR → executable.
- Paper II (Regulator) provides the control flow model: promoters and enhancers become the conditional logic that determines when edits are expressed.
- Paper III (RNAControl) provides the signaling model: non-coding RNAs become the middleware that coordinates between components.
- Paper IV (Structure) provides the memory model: genomic organization determines which regions are accessible to editing tools.
- Paper V (Repeat) provides the self-modification model: transposons are nature's CRISPR, and DNA-Lang's edit operations are their engineered counterpart.
- Paper VI (State) provides the state model: epigenetic marks determine the runtime accessibility that constrains where edits can be made.
- Paper VII (Program) provides the orchestration model: developmental programs show how to compose all previous types into coherent organism-level outcomes.

14.3 Limitations and Future Work

The current DNA-Lang implementation is a reference prototype. Key directions for future work include:

1. **Dependent types for biological constraints.** The current refinement type system is first-order. Dependent types would enable constraints like "the guide targets a sequence within the coding region of the declared target gene"—constraints that refer to the values of other bindings.
2. **Effect system for wet-lab operations.** Assay execution, cell culture, and sequencing are side effects. An effect system would enable the compiler to reason about which operations are pure (design, type checking) and which require lab resources.
3. **Multi-target programs.** The current syntax supports single-target programs. Combinatorial libraries (saturating mutagenesis, multiplexed CRISPR) require array types and iteration constructs.

4. **Temporal types for longitudinal experiments.** Cell culture experiments unfold over time. Temporal types would enable the compiler to verify that measurements are taken at the correct time points relative to treatment.

5. **Real backend integration.** The current implementation uses simulated functions. Connecting to real SBOL generators, Benchling APIs, AlphaFold endpoints, and LIMS systems would make DNA-Lang a practical tool.

6. **Formal verification.** The proofs sketched in Section 12 should be mechanized in a proof assistant (Coq, Lean, Agda) to provide machine-checked guarantees.

14.4 Conclusion

Biology is computation. The genome computes using functions (proteins), control flow (regulatory sequences), signals (RNA), memory (chromatin structure), self-modification (transposons), state (epigenetics), and orchestration (developmental programs). For decades, we have manipulated these computational primitives with untyped tools—passing sequences as strings, predictions as files, and protocols as prose.

DNA-Lang proposes that we can do better. By giving every biological entity a type, every editing operation a type signature, every AI prediction a confidence bound, and every experimental step a provenance record, we transform the biotech workflow from a fragile pipeline of string-passing into a robust, type-checked program.

The vision is not that DNA-Lang replaces wet-lab biology. The vision is that when a researcher writes `edit ko_edit : Knockout = knockout(pcsk9, top_guide)`, the type system guarantees that `pcsk9` is a valid target, `top_guide` is a compatible guide RNA, the resulting edit will be a knockout (not a base edit or insertion), and the compliance artifact will record every decision that led to this line of code.

Type safety is not just a programming language property. In biological programming, type safety is patient safety.

References

[1] M. Galdzicki *et al.*, "The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology," *Nature Biotechnology*, vol. 32, pp. 545–550, 2014.

[2] J. A. Doudna and E. Charpentier, "The new frontier of genome engineering with CRISPR-Cas9," *Science*, vol. 346, no. 6213, p. 1258096, 2014.

[3] J. Jumper *et al.*, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, pp. 583–589, 2021.

[4] A. M. Bellinger *et al.*, "In vivo base editing of PCSK9 as a therapeutic alternative to statins," *Nature*, vol. 628, pp. 550–556, 2024.

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

[6] P. Wadler, "Propositions as types," *Communications of the ACM*, vol. 58, no. 12, pp. 75–84, 2015.

[7] L. Cong *et al.*, "Multiplex genome engineering using CRISPR/Cas systems," *Science*, vol. 339, no. 6121, pp. 819–823, 2013.

[8] A. V. Anzalone *et al.*, "Search-and-replace genome editing without double-strand breaks or donor DNA," *Nature*, vol. 576, pp. 149–157, 2019.

[9] N. M. Gaudelli *et al.*, "Programmable base editing of A·T to G·C in genomic DNA without DNA cleavage," *Nature*, vol. 551, pp. 464–471, 2017.

[10] J. Abramson *et al.*, "Accurate structure prediction of biomolecular interactions with AlphaFold 3," *Nature*, vol. 630, pp. 493–500, 2024.

[11] P. M. Rondon, M. Kawaguchi, and R. Jhala, "Liquid types," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 159–169, 2008.

[12] J. A. McLaughlin *et al.*, "The Synthetic Biology Open Language (SBOL) Version 3: Simplified data exchange for bioengineering," *Frontiers in Bioengineering and Biotechnology*, vol. 8, p. 1009, 2020.

[13] M. Long, "Coding Sequences as Executable Functions: The Central Dogma as a Compilation Pipeline," *GrokRxiv*, 2026.

[14] M. Long, "Regulatory Sequences as Control Flow: Promoters, Enhancers, and the Logic of Gene Expression," *GrokRxiv*, 2026.

[15] M. Long, "Non-coding RNA as Signals and Middleware: miRNA, siRNA, and lncRNA as Biological Message Passing," *GrokRxiv*, 2026.

[16] M. Long, "Structural DNA as Memory Layout: Telomeres, Centromeres, and Topological Domains," *GrokRxiv*, 2026.

[17] M. Long, "Repetitive Elements as Self-Modifying Code: Transposons and Genomic Plasticity," *GrokRxiv*, 2026.

[18] M. Long, "Epigenetic Marks as Runtime State: DNA Methylation and Histone Modifications," *GrokRxiv*, 2026.

[19] M. Long, "Developmental Programs as Orchestration: Hox Genes, Morphogens, and Cell-Fate Networks," *GrokRxiv*, 2026.