Crafting a REPL for HOT Compiled Execution

MARTIN ELSMAN, University of Copenhagen, Denmark

A read-eval-print loop (REPL) supports interactive programming by accepting declarations, evaluating them, and printing their results in a persistent session. Incremental compilation provides a foundation for native execution by compiling successive program units using information retained from earlier units.

This compilation interface alone does not provide an interactive execution environment. Newly generated code must be linked to values already present in the session, and results must be printed even when their native representations omit runtime type information.

We present a REPL design for the optimising Standard ML compiler MLKit that combines its existing incremental compilation framework with standard dynamic linking and type-indexed pretty-printing. Each interaction forms a compilation unit. A compiler process generates shared libraries, and a persistent evaluation process loads and executes them; UNIX named pipes connect the processes. For printing, the compiler supplies type and representation descriptions that guide inspection of native values, including records without runtime type tags and datatypes represented without a separate constructor object. The design integrates interactive declarations and separately compiled libraries within the same compilation and loading infrastructure.

CCS Concepts: • **Software and its engineering → Functional languages**; *Runtime environments.*

Additional Key Words and Phrases: Compilation environments, Evaluation environments, Functional languages

1 INTRODUCTION

A read-eval-print loop (REPL) accepts a declaration, evaluates it, prints the resulting bindings, and continues in the extended programming session. In Standard ML, declarations introduce values, types, and modules. Native execution in such a session requires both the compiler information needed by subsequent declarations and the runtime values those declarations may use.

Incremental compilation processes a program one unit at a time, using information obtained from previously compiled units. Elsman's framework for cut-off incremental recompilation and inter-module optimisation [7] provides this foundation for MLKit, an instance of a compiler for a *Higher-Order and Typed* (HOT) language. Here, cut-off means that a change to one unit need not force recompilation of every dependent unit: reuse depends on the compilation assumptions that the unit requires.

The compilation framework explains how information passes between compiled units, but an interactive environment must also connect those units during execution. Each new declaration must execute in a process where earlier values remain available, and its results must become accessible both to subsequent declarations and to the user. The design must therefore coordinate the retained compiler information with the code and values held by the running session.

Inspection of results presents a further problem. MLKit uses *untagged records*, whose fields carry no runtime type tags, and *unboxed datatypes*, whose values may omit a separate constructor object [3]. Such values do not themselves contain all the information needed to print them. Providing a REPL therefore requires a connection between the compiler's knowledge of representations and the values produced by native execution.

We address these requirements by treating each interactive toplevel declaration as a compilation unit and connecting incremental compilation to a persistent evaluation process. Compiling a declaration produces native code and compiler information for later interactions. This adapts MLKit's existing compilation infrastructure to the successive declarations of an interactive session. The REPL works for both MLKit and ReML [9], a language based on MLKit that extends Standard

DIKU Technical Report, 2026-09-19.

Author's address: Martin Elsman, Department of Computer Science, University of Copenhagen, Universitetsparken 5, Copenhagen, Denmark, DK-2100, mael@di.ku.dk.

ML with explicit region and effect annotations and effect constraints. Both use the incremental compilation and native execution infrastructure described here.

The design uses two cooperating processes. The compiler process reads declarations, checks their types, and compiles them into shared libraries, which are binary objects that can be loaded into a running process. The evaluation process loads these libraries with the system's dynamic linker and executes their initialisation code. It retains the runtime values produced during the session. UNIX named pipes carry commands and replies between the processes.

To print a value, the compiler supplies a description of its type and representation to the evaluation process. The evaluation process uses this description to inspect and format the native value. We refer to this method as *type-indexed pretty-printing*. It makes the compiler's representation information available where the value resides.

The same design supports library loading. An ML Basis (MLB) file specifies source files and their dependencies. The REPL invokes MLKit's compilation manager on such a file, loads the resulting code, and makes its exported bindings available to subsequent interactions. This is also how a session acquires the Standard ML Basis Library.

The paper makes the following contributions:

(1) We show how to organise interactive declarations as compilation units within MLKit's existing incremental compilation framework, retaining the compilation information needed by later interactions.
(2) We give a formal development that states precisely the assumed compiler properties under which incremental compiled execution corresponds to the source model and preserves uniqueness of linked names. We mechanise a simplified instance of this development in Rocq.
(3) We present a two-process REPL design that connects incremental native compilation to immediate execution through shared libraries, standard dynamic linking, and a command protocol.
(4) We describe type-indexed pretty-printing of native values, including untagged records and unboxed datatypes, using descriptions supplied by the compiler.

Interactive use also places demands on latency. Launching a session, loading a library, and compiling and executing a small declaration must be considered separately: fast native execution alone does not establish that a REPL is responsive. Section 6 reports regression tests and a loader microbenchmark; complete interaction latency remains to be measured.

Section 2 specifies the behaviour of a session in terms of static and dynamic bases. Section 3 presents the design based on incremental compilation and states its compiler properties and simulation theorems. Section 4 describes their mechanisation in Rocq. Section 5 describes the implementation, including type-indexed pretty-printing in Section 5.6. Section 6 reports functional tests and an incremental-linking experiment. Section 7 discusses related work, and Section 8 concludes with directions for future work.

2 A MODEL OF INTERACTIVE EXECUTION

Assume a language with toplevel declarations d. A *static basis* B_s contains the information used to check declarations; a *dynamic basis* B_d contains their runtime bindings. A state s records mutable storage and any other state of evaluation. A session configuration is $\langle B_\mathrm{s}, B_\mathrm{d}, s\rangle$. We follow the organisation of programs in *The Definition of Standard ML* [16, Section 8, rules (187)–(189)], but we model the notion of fresh type names differently. Instead, we make use of Elsman's development of modules [4, Chapter 2], which models freshness of type names using existential quantification. This idea was independently developed by Russo [19]. For finite maps M and M', the *extension* of

M by M', written $M + M'$, is the finite map with domain $\mathrm{Dom}(M + M') = \mathrm{Dom}(M) \cup \mathrm{Dom}(M')$ and values

$$(M + M')(x) = \begin{cases} M'(x) & x \in \mathrm{Dom}(M') \\ M(x) & x \in \mathrm{Dom}(M) \setminus \mathrm{Dom}(M') \end{cases}$$

Basis extension $B + B'$ applies this operation componentwise to the identifier environments; B' is the declaration's contribution. New bindings shadow earlier bindings of the same identifier. Extension of a static/dynamic pair is componentwise:

$$(B_\mathrm{s}, B_\mathrm{d}) + (B'_\mathrm{s}, B'_\mathrm{d}) = (B_\mathrm{s} + B'_\mathrm{s}, B_\mathrm{d} + B'_\mathrm{d})$$

We assume a static-semantics judgement $B_\mathrm{s} \vdash_\mathrm{stat} d \Rightarrow \exists N.B'_\mathrm{s}$. The finite set N binds the generated type names in the resulting basis contribution. Results are identified up to consistent, capture-avoiding renaming of these bound names. Write $\mathsf{tynames}(X)$ for the free type names of X, taking unions for tuples. Source declarations contain type identifiers, not semantic type names: $\mathsf{tynames}(d) = \emptyset$. To use a result, open its binder with type names fresh for the input static basis. The declaration rules below rebind these names in their results. A session opens the resulting package before extending its bases; subsequent openings must respect the identities of names already in scope.

We also assume an evaluation judgement $s, B_\mathrm{d} \vdash_\mathrm{dyn} d \Rightarrow r, s'$. The result r is either $\mathsf{ok}(B'_\mathrm{d})$ or $\mathsf{exn}(v)$, where v is an exception value. We write $B_\mathrm{s} \vdash_\mathrm{stat} d \Rightarrow \mathsf{error}$ for rejection. The same d denotes the declaration in both judgements, with any erasure required by the language's dynamic semantics left implicit. Type names do not occur in the dynamic basis or state, so opening the static result does not rename dynamic objects.

Write $B = (B_\mathrm{s}, B_\mathrm{d})$ and let $\mathbf{0}$ denote the pair of empty basis contributions, so $B + \mathbf{0} = B$. A declaration is processed by the judgement

$$s, B \vdash d \Rightarrow \exists N.(B', r, s')$$

Here B' is its basis contribution, r is done, error, or $\mathsf{exn}(v)$, and s' is the resulting state. The binder scopes over the entire result; type names occur only in the static component of B'. Processing produces a contribution without extending the input basis. The rules are

$$\frac{B_\mathrm{s} \vdash_\mathrm{stat} d \Rightarrow \mathsf{error}}{s, (B_\mathrm{s}, B_\mathrm{d}) \vdash d \Rightarrow \exists \emptyset.(\mathbf{0}, \mathsf{error}, s)} \tag{1}$$

$$\frac{B_\mathrm{s} \vdash_\mathrm{stat} d \Rightarrow \exists N.B'_\mathrm{s} \qquad N \cap \mathsf{tynames}(B_\mathrm{s}) = \emptyset \qquad s, B_\mathrm{d} \vdash_\mathrm{dyn} d \Rightarrow \mathsf{ok}(B'_\mathrm{d}), s'}{s, (B_\mathrm{s}, B_\mathrm{d}) \vdash d \Rightarrow \exists N.((B'_\mathrm{s}, B'_\mathrm{d}), \mathsf{done}, s')} \tag{2}$$

$$\frac{B_\mathrm{s} \vdash_\mathrm{stat} d \Rightarrow \exists N.B'_\mathrm{s} \qquad N \cap \mathsf{tynames}(B_\mathrm{s}) = \emptyset \qquad s, B_\mathrm{d} \vdash_\mathrm{dyn} d \Rightarrow \mathsf{exn}(v), s'}{s, (B_\mathrm{s}, B_\mathrm{d}) \vdash d \Rightarrow \exists \emptyset.(\mathbf{0}, \mathsf{exn}(v), s')} \tag{3}$$

Thus, rejection leaves the state unchanged. An exception discards both basis contributions and their generated type names, but preserves state effects. Normal completion retains the generated names under the result binder. The rules compose static checking and dynamic evaluation of the same declaration, with type information erased for evaluation.

Let $q = \langle B_s, B_d, s\rangle$ range over session configurations. A transition $q \xrightarrow{d/r} q'$ opens a declaration result freshly and applies its contribution:

$$\frac{s, (B_s, B_d) \vdash d \Rightarrow \exists N.((B'_s, B'_d), r, s') \qquad N \cap \mathsf{tynames}(B_s) = \emptyset}{\langle B_s, B_d, s\rangle \xrightarrow{d/r} \langle B_s + B'_s, B_d + B'_d, s'\rangle}$$

Every occurrence of the contribution in this rule uses the same fresh opening. Subsequent transitions keep these names fixed. Empty contributions make this rule apply uniformly to rejection and exceptions. The reported outcome records completion; the resulting bases retain the bindings available for inspection. Printing is outside the model. Successive interactions compose these transitions from the initial bases and state. The rules describe terminating interactions and abstract from parsing and external failures.

3 DESIGN BASED ON INCREMENTAL COMPILATION

The model specifies source-language behaviour. The design implements that behaviour by compiling declarations and linking their object code into a persistent process. We introduce a *compilation basis*: the collection of information retained across compiler phases for compiling subsequent units. An *export basis* is the information contributed by a compiled unit. These are compiler objects, distinct from the static and dynamic bases of Section 2.

We use MLKit's incremental compilation framework [7, Section 3]. We first specify environments, compilation bases, and the composition of translation phases.

3.1 Compilation Bases

A compilation basis is an environment or a finite, nested tuple of compilation bases. Its shape specifies the domain and codomain of each constituent environment. Two bases are compatible when they have the same shape and corresponding environments have the same domains and codomains. For compatible bases, extension is defined componentwise. In particular,

$$(E_1, E_2) + (F_1, F_2) = (E_1 + F_1, E_2 + F_2)$$

At an environment leaf, extension is the operation defined above. Associativity follows by induction on the common shape, using associativity of environment extension at each leaf. A tuple of environments is kept distinct from a finite map on a product domain; no isomorphism between these types is assumed.

3.2 Translation Phases

A *name* identifies a source binding or a compiler-generated entity, such as an intermediate variable or a code label. For an object X, let $\mathsf{names}(X)$ denote its free names; for a tuple, this is the union of the free names of its components. A *base translation phase* relates a source program p, a target program p', and input and export environments E and E' through a judgement

$$E \vdash p \Rightarrow \exists N.(E', p')$$

Here N is the set of names generated by the phase. The prefix $\exists N$ binds these names in the result; bound names may be renamed consistently without capture. The judgement expresses translation of p under assumptions E, producing p' and export environment E'.

The first compiler phase is *elaboration*. Besides performing the checks of the static semantics, it returns a typed intermediate declaration d^{t}, containing the type information needed by later

phases:

$$B_{\mathrm{s}} \vdash_{\mathrm{elab}} d \Rightarrow \exists N.(B'_{\mathrm{s}}, d^{\mathrm{t}})$$

The binder scopes jointly over the basis contribution and typed declaration: occurrences of a generated type name in both denote the same type. Unlike the source declaration d, the typed intermediate declaration d^{t} may contain semantic type names in its type annotations. Erasing the typed declaration from the result recovers the static-semantics judgement. More precisely, up to bound-name renaming,

$$B_{\mathrm{s}} \vdash_{\mathrm{stat}} d \Rightarrow \exists N.B'_{\mathrm{s}} \quad \Longleftrightarrow \quad \exists d^{\mathrm{t}}.\, B_{\mathrm{s}} \vdash_{\mathrm{elab}} d \Rightarrow \exists N.(B'_{\mathrm{s}}, d^{\mathrm{t}})$$

Here the two packages use a common fresh representative; elaboration rejects exactly when static checking rejects. In the complete compilation basis, the static basis is the component used by this first phase. Later phases consume d^{t} and their respective basis components. Type names and generated runtime symbols are distinct sorts of names; only runtime symbols occur in native object interfaces.

Two translation phases may be fused using the following rule:

$$\frac{\begin{array}{cc} E \vdash p \Rightarrow \exists N.(E', p') & F \vdash p' \Rightarrow \exists N'.(F', p'') \\ (N \cup N') \cap \mathrm{names}(E, F) = \emptyset & N' \cap (N \cup \mathrm{names}(E', p')) = \emptyset \end{array}}{(E, F) \vdash p \Rightarrow \exists (N \cup N').((E', F'), p'')} \tag{4}$$

Here E, E', F, and F' are bases, N and N' are names sets, p' is an intermediate program, and p and p'' are the source program and the target program, respectively.

3.3 Propagation of Compile-Time Information

In the REPL implementation, the compilation basis connects the source-level identifiers of earlier interactions to the information required by later compiler phases. For example, a request to print a value needs both its type description and the code label through which the evaluation process can retrieve it. The compiler obtains these from the stored compilation information. Thus, retaining only the types reported to the user would not provide the interface needed by the REPL.

The underlying framework permits information containing free names to cross unit boundaries. Reuse requires conditions on name propagation, restriction to required assumptions, extension by additional assumptions, and determinacy [7, Sections 3–5].

The implementation reconstructs a basis before compiling each new interaction and saves the resulting export basis alongside the generated code. Section 5 describes this organisation.

3.4 Object Interfaces and Linking

Cardelli models separately compiled fragments by *linksets*, with explicit interfaces, compatibility checks, and linking steps based on substitution [2, Sections 5–6]. We adapt this organisation to native object files and incremental loading. Substitution here resolves symbolic references to addresses; it does not evaluate or copy the computations that produced existing values. Our model specialises to objects whose external dependencies are already loaded. Recursion within an object is handled by its internal definitions. Cardelli's results for his typed fragment calculus do not by themselves establish correctness of this native-code adaptation.

An *interface* is a finite map from symbol names to representation contracts κ, describing the kind and layout of a datum or the calling convention of a function. Contract equality is assumed. An object body is $o = (I_o, E_o, b_o)$, where I_o is its import interface, E_o its export interface, and b_o its code and data image, including an initialisation entry. Define

$$\mathrm{imp}(o) = \mathrm{Dom}(I_o) \qquad \mathrm{exp}(o) = \mathrm{Dom}(E_o)$$

Imports are externally supplied symbols; exports are defined by the object. These sets are disjoint. All external symbolic references in b_o occur in I_o, and each name in E_o has a definition in b_o. References between definitions of the same object are internal. Write $\mathsf{wf}(o)$ for these conditions together with consistency of the image with its contracts. Establishing this consistency is a compiler obligation; the contracts need not be stored in a native object file.

An *object file* packages its body with an existential name binder:

$$O = \exists N.o = \exists N.(I_o, E_o, b_o)$$

The finite set N binds the generated names throughout the body, including export declarations and their occurrences in code and data. We require $\mathrm{exp}(o) \subseteq N$ and $N \cap \mathrm{imp}(o) = \emptyset$. Imported symbol names are free: they identify definitions supplied by other objects.

Object files are considered equal up to consistent renaming of their bound names. If π is a finite permutation of names fixing every free name of $\exists N.o$, then

$$\exists N.o \equiv_\alpha \exists \pi(N).\pi(o)$$

Here π acts on binding declarations and occurrences together, with capture avoided. Thus, two object files may use the same bound name in their stored representations; opening their binders with disjoint fresh names separates the definitions before linking. If a compiled client already refers to an export, renaming that shared name requires a common enclosing binder over provider and client. Freshening the provider alone would leave the client referring to the old name.

A loaded collection is $L = (\Sigma_L, A_L, P_L)$, where Σ_L records export contracts, A_L maps their names to allocated addresses, and P_L is the ordered list of loaded images. We require $\mathrm{Dom}(\Sigma_L) = \mathrm{Dom}(A_L)$, distinct exported names, and resolution of all external references in P_L. The initial collection includes the runtime's exported symbols. An object is *compatible* with L, written $L \bowtie o$, when

$$\begin{aligned} &\mathrm{imp}(o) \subseteq \mathrm{Dom}(\Sigma_L) \quad \wedge \quad \mathrm{exp}(o) \cap \mathrm{Dom}(\Sigma_L) = \emptyset \\ &\wedge \quad (\forall n \in \mathrm{imp}(o).\ I_o(n) = \Sigma_L(n)) \end{aligned}$$

The export-disjointness condition is essential: a generated symbol may be defined only once in a loaded collection. To link a packaged object $\exists N.o$, first choose an alpha-equivalent representative with N fresh for the collection and the importing context. The operation $L \cdot o$ below acts on that opened body. Its import check resolves free names; its export check prevents a second definition of an existing name.

Let θ be a finite substitution from symbols to addresses. Write $L \vdash I \rightsquigarrow \theta$ for resolution of an import interface. For a finite map I, $I \setminus \{n\}$ removes its entry for n; $\uplus$ denotes disjoint union of maps. Resolution is inductively defined:

$$\frac{}{L \vdash \emptyset \rightsquigarrow \emptyset} \qquad (5)$$

$$\frac{n \in \mathrm{Dom}(I) \qquad I(n) = \Sigma_L(n) \qquad A_L(n) = a \qquad L \vdash I \setminus \{n\} \rightsquigarrow \theta}{L \vdash I \rightsquigarrow \theta \uplus \{n \mapsto a\}} \qquad (6)$$

Induction on these rules gives $\mathrm{Dom}(\theta) = \mathrm{Dom}(I)$ and $\theta(n) = A_L(n)$ for each import n. Each recursive premise removes one entry. When the import contracts agree with Σ_L, induction on the size of I constructs a resolution derivation. The resulting substitution is independent of the order of resolution.

Let α assign the exported definitions of o fresh addresses in a valid layout of b_o, with $\mathrm{Dom}(\alpha) = \mathrm{exp}(o)$. Write $\mathsf{layout}(L, o, \alpha)$ for this allocation condition. The relocated image $b_o[\theta \uplus \alpha]$ resolves both external imports and references to the object's exported definitions; local bindings are respected. Write @ for list concatenation. Linking and loading are specified by

$$\frac{\mathsf{wf}(o) \qquad L \bowtie o \qquad L \vdash I_o \rightsquigarrow \theta \qquad \mathsf{layout}(L, o, \alpha)}{L \cdot o = (\Sigma_L \uplus E_o,\ A_L \uplus \alpha,\ P_L @ [b_o[\theta \uplus \alpha]])} \qquad (7)$$

In particular, the rule gives

$$\mathrm{Dom}(\Sigma_{L\cdot o}) = \mathrm{Dom}(\Sigma_L) \mathbin{\dot\cup} \exp(o)$$

where $\dot\cup$ denotes disjoint union of sets. Starting from distinct runtime exports, induction on linking derivations therefore preserves uniqueness of loaded symbol definitions. This argument also covers objects retained after exceptions. Alpha-renaming permits a fresh instance of an object to be linked; it does not identify that instance with an earlier one or justify executing its initialiser twice.

This operation preserves all earlier symbol bindings and images. It allocates and relocates the new image; the mutable state h changes when its initialisation entry is executed. Allocation is abstracted by layout, whose existence and correctness must be supplied by the loader implementation. Resource failures are outside the model.

3.5 Incremental Linking and Execution

Let C denote a compilation basis for the complete pipeline, d a toplevel declaration, and o its generated object code. Instantiating the translation judgement gives

$$C \vdash d \Rightarrow \exists N.(C', o)$$

where C' is the export basis and N contains the generated names. Here $o = (I_o, E_o, b_o)$ is an object body. The binder scopes over the *pair* (C', o), so that a name exported in C' denotes the same entity as its definition in o. Projecting away C' yields the packaged object file $\exists N.o$. In particular,

$$\exists N.(C', o) \equiv_\alpha \exists \pi(N).(\pi(C'), \pi(o))$$

for a permutation fixing the free names of the pair. Renaming the object without the corresponding export basis would break their agreement. The compiler must produce $\mathsf{wf}(o)$ with $\exp(o) \subseteq N$. Its import interface must agree with the contracts for the corresponding symbols in C, and those symbols must be available in the loaded collection. These are requirements on the compilation interface that connect name propagation to linking.

A concrete session is $K = \langle C, L, h\rangle$, where L records loaded objects and their symbol bindings, and h is the runtime state. Write $\mathsf{fresh}(N; K, d)$ when the names in N are fresh with respect to C, L, h, and d, including the names allocated by all retained objects. Before using a compilation result, we open $\exists N.(C', o)$ with such a fresh choice, applying the same capture-avoiding renaming to C' and o. Imported names retain their existing identities. The existential premise in each rule below is taken up to $\equiv_\alpha$, and its freshness premise selects one jointly renamed representative for every occurrence of N, C', and o in that rule. Once loaded, their names remain fixed across interactions, including after exceptional completion.

The linking premise below is derived using rule (7). Source-level shadowing is represented in $C + C'$; disjoint object exports ensure that it does not rebind symbols in previously compiled code. The judgement

$$(L \cdot o, h) \Downarrow \rho, h'$$

executes only the initialisation entry of the newly appended image. Its outcome is done or $\mathsf{raised}(w)$ for a native exception value w. We also admit the empty object ϵ, with empty import and export interfaces, no code or data, and a trivial initialisation that returns done without changing h. It introduces no symbol definitions and requires no relocation. Earlier initialisation computations are not rerun. Write $C \vdash d \Rightarrow \mathsf{error}$ for rejection by elaboration; this judgement does not describe compiler or linker failures.

A compiled interaction is a transition $K \xrightarrow{d/\widehat{r}}_{\mathrm{c}} K'$, where the reported outcome $\widehat{r}$ is error, done, or raised(w). Its rules correspond to rules (1)–(3):

$$\frac{C \vdash d \Rightarrow \mathsf{error}}{\langle C, L, h\rangle \xrightarrow{d/\mathsf{error}}_{\mathrm{c}} \langle C, L, h\rangle} \tag{8}$$

$$\frac{\begin{array}{c} C \vdash d \Rightarrow \exists N.(C', o) \qquad \mathsf{fresh}(N; \langle C, L, h\rangle, d) \\ L' = L \cdot o \qquad (L', h) \Downarrow \mathsf{done}, h' \end{array}}{\langle C, L, h\rangle \xrightarrow{d/\mathsf{done}}_{\mathrm{c}} \langle C + C', L', h'\rangle} \tag{9}$$

$$\frac{\begin{array}{c} C \vdash d \Rightarrow \exists N.(C', o) \qquad \mathsf{fresh}(N; \langle C, L, h\rangle, d) \\ L' = L \cdot o \qquad (L', h) \Downarrow \mathsf{raised}(w), h' \end{array}}{\langle C, L, h\rangle \xrightarrow{d/\mathsf{raised}(w)}_{\mathrm{c}} \langle C, L', h'\rangle} \tag{10}$$

The premise $L' = L \cdot o$ requires a derivation of rule (7), including interface compatibility and import resolution. In rule (9), extension and linking use the same fresh names in C' and o. Rule (10) retains the previous compilation basis but keeps the object loaded: an earlier side effect may retain a function defined by it. The freshness condition of later interactions therefore includes this object, although its export basis was not installed.

3.6 Correspondence with the Model

The intended relationship is expressed by a representation relation

$$\mathcal{R}(C, L, h; B_{\mathrm{s}}, B_{\mathrm{d}}, s)$$

This relation requires the source-level information in C to agree with B_{s}, the information in C together with the symbols in L to represent the bindings in B_{d}, and the runtime state h to represent s. It may relate native code to semantic function values and native addresses to semantic locations; it need not be a literal equality of objects. Extra loaded objects need not introduce source-level bindings.

For configurations $K = \langle C, L, h\rangle$ and $q = \langle B_{\mathrm{s}}, B_{\mathrm{d}}, s\rangle$, abbreviate this relation as $\mathcal{R}(K; q)$. Write $\widehat{r} \simeq_{K', q'} r$ for correspondence of reported outcomes in the resulting configurations. Rejections correspond to rejections; done corresponds to done, with exported bindings related through $\mathcal{R}$; and raised(w) corresponds to exn(v) when w represents v. This outcome relation, like $\mathcal{R}$, must be instantiated for the compiler's representations.

We attach correctness to a compilation result, before considering a session transition. This follows the organisation of translation phases and jointly bound export bases in the incremental compilation framework [7, Section 3]. That framework separates structural properties of translation, such as name propagation and closure under enrichment and restriction, from correctness of the individual transformations. Here we add a semantic property; those structural properties alone do not establish it.

Write $\mathsf{V}(C, d, \exists N.(C', o))$ for validity of a fixed result. The definition quantifies over every $K = \langle C, L, h\rangle$ and $q = \langle B_{\mathrm{s}}, B_{\mathrm{d}}, s\rangle$ satisfying $\mathcal{R}(K; q)$, and every jointly fresh opening of the result. For each such opening, $\mathsf{wf}(o)$ and $L \bowtie o$ must hold. Assuming the loader supplies a valid layout, let $L' = L \cdot o$. For every elaboration $B_{\mathrm{s}} \vdash_{\mathrm{stat}} d \Rightarrow \exists N_{\mathrm{t}}.B'_{\mathrm{s}}$, open the type names freshly as in Section 2. The representation relation must respect renaming of these names; choose their identities

consistently with the elaboration component of the compiled result. Validity then requires the following two clauses.

If $s, B_{\mathrm{d}} \vdash_{\mathrm{dyn}} d \Rightarrow \mathsf{ok}(B'_{\mathrm{d}}), s'$, there exists h' such that

$$(L', h) \Downarrow \mathsf{done}, h'$$
$$\wedge \quad \mathcal{R}(C + C', L', h'; B_{\mathrm{s}} + B'_{\mathrm{s}}, B_{\mathrm{d}} + B'_{\mathrm{d}}, s')$$

If $s, B_{\mathrm{d}} \vdash_{\mathrm{dyn}} d \Rightarrow \mathsf{exn}(v), s'$, there exist w and h' such that

$$(L', h) \Downarrow \mathsf{raised}(w), h'$$
$$\wedge \quad \mathcal{R}(C, L', h'; B_{\mathrm{s}}, B_{\mathrm{d}}, s')$$

with w representing v in these resulting configurations. Thus the same compiled result works in every represented runtime realisation of its input basis; it is not selected separately for each evaluation. The definition is independent of the compilation relation. Compiler correctness is the obligation

$$C \vdash d \Rightarrow \exists N.(C', o) \quad \Longrightarrow \quad \mathsf{V}(C, d, \exists N.(C', o))$$

Joint renaming must preserve validity. In particular, $C+C'$ must retain the dependencies of exported information even when source names are shadowed. Neither this property nor equivariance of native code follows merely from writing an existential binder.

Correctness of existing derivations does not ensure that a derivation exists. Define $\mathsf{A}(C, B_{\mathrm{s}})$ to express agreement of the compilation basis with the static basis, and require $\mathcal{R}(K; q)$ to imply this agreement. Let P range over result packages $\exists N.(C', o)$. Compilation existence is the separate static obligation

$$\mathsf{A}(C, B_{\mathrm{s}}) \quad \wedge \quad B_{\mathrm{s}} \vdash_{\mathrm{stat}} d \Rightarrow \exists N_{\mathrm{t}}.B'_{\mathrm{s}}$$
$$\Longrightarrow \quad \exists P.\, C \vdash d \Rightarrow P$$

Under $\mathsf{A}(C, B_{\mathrm{s}})$, static rejection must likewise imply $C \vdash d \Rightarrow \mathsf{error}$. These obligations do not assume termination of evaluation.

Let $\mathsf{U}(K)$ mean that no runtime symbol is defined twice among the images retained in K, including the initial runtime and objects retained after exceptions. Section 3.4 establishes preservation of this invariant by linking: each new export is disjoint from the preceding exports.

We name the required properties below. Each defining formula is universally closed over its free variables; P ranges over compilation result packages and Q over static result packages. Write $\mathsf{V}(C, d, P)$ using the validity predicate defined above.

$$\mathsf{CC} \equiv (C \vdash d \Rightarrow P) \Longrightarrow \mathsf{V}(C, d, P)$$

$$\mathsf{CE} \equiv \big(\mathsf{A}(C, B_{\mathrm{s}}) \quad \wedge \quad B_{\mathrm{s}} \vdash_{\mathrm{stat}} d \Rightarrow Q\big) \Longrightarrow \exists P.\, C \vdash d \Rightarrow P$$

$$\mathsf{CR} \equiv \big(\mathsf{A}(C, B_{\mathrm{s}}) \quad \wedge \quad B_{\mathrm{s}} \vdash_{\mathrm{stat}} d \Rightarrow \mathsf{error}\big) \Longrightarrow C \vdash d \Rightarrow \mathsf{error}$$

$$\mathsf{RA} \equiv \mathcal{R}(C, L, h; B_{\mathrm{s}}, B_{\mathrm{d}}, s) \Longrightarrow \mathsf{A}(C, B_{\mathrm{s}})$$

The linking operation is the relation defined by rule (7). Its remaining allocation requirement is

$$\mathsf{LA} \equiv \forall K = \langle C, L, h\rangle, o.\, \big(\mathsf{U}(K) \quad \wedge \quad \mathsf{wf}(o) \quad \wedge \quad L \bowtie o\big) \Longrightarrow \exists \alpha.\, \mathsf{layout}(L, o, \alpha)$$

For renaming, π ranges over finite, sort-preserving permutations, acting consistently on configurations and result packages. The following property makes validity independent of bound representatives and representation invariant under a common renaming:

$$\mathsf{RN} \equiv \begin{array}{c} \big(\forall C, d, P, P'.\, P \equiv_{\alpha} P' \Longrightarrow (\mathsf{V}(C, d, P) \Longleftrightarrow \mathsf{V}(C, d, P'))\big) \\ \wedge \quad \big(\forall \pi, K, q.\, \mathcal{R}(K; q) \Longleftrightarrow \mathcal{R}(\pi K; \pi q)\big) \end{array}$$

Joint fresh opening within V includes the consistent choice of source type names specified above. Abbreviate the conjunction of these properties by

$$\mathsf{H} \equiv \mathsf{CC} \quad \wedge \quad \mathsf{CE} \quad \wedge \quad \mathsf{CR} \quad \wedge \quad \mathsf{RA} \quad \wedge \quad \mathsf{LA} \quad \wedge \quad \mathsf{RN}$$

Theorem 1 (Single-interaction simulation). *Assume* H. *Then for all q, d, r, q', and K such that (1) $q \xrightarrow{d/r} q'$ and (2) $\mathcal{R}(K; q)$ and (3) $\mathsf{U}(K)$, there exist K' and $\widehat{r}$ such that $K \xrightarrow{d/\widehat{r}}_{\mathrm{c}} K'$ and $\mathcal{R}(K'; q')$ and $\widehat{r} \simeq_{K',q'} r$ and $\mathsf{U}(K')$.*

Proof. Invert the source session update to obtain a declaration result and proceed by cases on its derivation. Rejection uses agreement on static rejection and leaves both configurations unchanged. In the other cases, static agreement and compilation existence supply a compilation result. Open it consistently with the source result. Validity supplies object compatibility and the appropriate normal or exceptional execution, with related resulting configurations and outcomes. Import resolution and the assumed layout give a linking derivation; disjoint exports preserve U. Apply rule (9) or rule (10), respectively. The exceptional case retains the new object, so its symbols remain covered by the invariant. □

This factoring leaves the session induction independent of compiler passes. To establish validity compositionally, each pass must preserve an appropriate relation between its input and output representations, including their bases and shared names. Proving those pass-specific properties for MLKit remains an obligation, rather than a consequence of phase composition alone.

3.7 Sequences of Interactions

A REPL session repeatedly uses the bases and runtime state produced by earlier interactions. We therefore lift the single-interaction result to finite sequences, preserving correspondence and name uniqueness throughout the session.

Let D be a finite list of declarations and O a list of reported outcomes. Write $[]$ for the empty list and $x :: X$ for list construction. For either the source transition relation (subscript s, omitted above) or the compiled relation (subscript c), define $Q \overset{D/O}{\Longrightarrow}_{\chi} Q'$ by the following rules, where $\chi \in \{\mathrm{s}, \mathrm{c}\}$ and Q ranges over the corresponding configurations:

$$\frac{}{Q \overset{[]/[]}{\Longrightarrow}_{\chi} Q} \tag{11}$$

$$\frac{Q \xrightarrow{d/r}_{\chi} Q_1 \qquad Q_1 \overset{D/O}{\Longrightarrow}_{\chi} Q_2}{Q \overset{(d::D)/(r::O)}{\Longrightarrow}_{\chi} Q_2} \tag{12}$$

Outcome correspondence for sessions is understood at each step: for source states q_i and compiled states K_i along the two sessions, the outcomes r_i and $\widehat{r}_i$ satisfy $\widehat{r}_i \simeq_{K_i,q_i} r_i$ in their resulting configurations. The lists need not be equal, since native exception values may differ from their semantic representations.

Theorem 2 (Finite-session simulation). *Assuming* H, *if $q \overset{D/O}{\Longrightarrow}_{\mathrm{s}} q'$ and $\mathcal{R}(K; q)$ and $\mathsf{U}(K)$, then there exist a compiled configuration K' and an outcome list $\widehat{O}$ such that $K \overset{D/\widehat{O}}{\Longrightarrow}_{\mathrm{c}} K'$ and $\mathcal{R}(K'; q')$ and $\mathsf{U}(K')$. The witnessing sessions have related configurations and corresponding outcomes after every interaction and* U *holds throughout the compiled session.*

Proof. Induct on the source session derivation. For rule (11), choose the initial compiled configuration and the empty outcome list. For rule (12), apply Theorem 1 to the first interaction. Its resulting representation relation and name invariant supply the hypotheses for the induction hypothesis on the remaining session. Prepend the compiled step and its outcome using rule (12). Fresh openings preserve the identities of names from the preceding session. □

Both theorems concern finite, terminating source derivations, including exceptional completion. They establish forward simulation, not termination of arbitrary programs or a converse simulation.

4 MECHANISATION IN ROCQ

A Rocq development mechanises a simplified instance of the preceding models. The development is available at https://github.com/melsman/repl-rocq. It separates name allocation and linking (`Names.v`) from interaction and session simulation (`Simulation.v`). External names are natural numbers; existentially bound names are represented by local indices. Opening an object replaces these indices with names above all previously loaded names. The development proves that references resolve, linking succeeds for well-formed objects, and linking preserves uniqueness of loaded names. An opening lemma shows that changing the fresh names consistently preserves external references. Allocated names are retained even after an initialiser raises an exception.

The source relation `process` returns a declaration result containing an optional pair of basis contributions, an outcome, and a store. Absence of contributions represents **0**. The function `apply_result` applies these contributions, and a single constructor defines the source session step. Rejection and exception lemmas establish that these outcomes contribute no bindings; rejection also preserves the store. The compiled interaction relation distinguishes rejection, normal completion, and exceptional completion. They follow the treatment of bases in Section 8 of the Definition [16]: an uncaught exception retains state effects without extending the static or dynamic basis. The compiled relation also retains the previous compilation basis and the newly loaded names. Basis extension and the representation relation are parameters; the operation modelling $C + C'$ receives the same fresh name base as object execution.

The predicate `correct_result` expresses declaration-level validity independently of the compilation relation. The `compiler_correct` parameter requires every successful compilation derivation to satisfy it in every represented runtime state. Static agreement, compilation existence, and agreement on rejection are separate parameters. Normal and exceptional execution lemmas are derived from validity, rather than assumed separately. The theorem `compiled_result_links` derives link progress and name uniqueness for each compiled result in a represented state with unique loaded names.

This instance uses a canonical fresh opening and checks reference scope; it abstracts from the paper's representation contracts and native layout. Static elaboration is a parameter returning an already opened basis contribution; existential type-name binding and the typed intermediate declaration are not yet represented explicitly in Rocq. Its opening lemma concerns references, not equivariance of a concrete compiler or its basis contribution. These remain obligations when instantiating the declaration-level property.

The two simulation theorems in `Simulation.v` have the following statements. Here `interpret` and `compiled` are the source and compiled step relations, `represents q k` is $\mathcal{R}(K; q)$, and `loaded k` is the retained name list. The compiler properties introduced above are enclosing Rocq section parameters, generalised as theorem arguments when the section closes.

```
Theorem step_simulation :
  forall q d a q', interpret q d a q' ->
  forall k, represents q k -> NoDup (loaded k) ->
  exists k', compiled k d a k' /\
    represents q' k' /\ NoDup (loaded k').
```

This is the abstract instance of Theorem 1. Its proof unwraps the source session update and invokes `declaration_simulation`, which constructs a matching compiled step and relates the applied declaration result to the target state.

The relation `session` is the inductive closure of a step relation over a list `trace` of declaration/outcome pairs, corresponding to the lists D and O in Section 3.7.

```
Theorem session_simulation :
  forall q trace q', session interpret q trace q' ->
  forall k, represents q k -> NoDup (loaded k) ->
  exists k', session compiled k trace k' /\
    represents q' k' /\ NoDup (loaded k').
```

This gives Theorem 2's final-state and trace conclusions in the abstract model. The proof follows the same induction on the source session, applying `step_simulation` at each step. The induction maintains the representation relation and name uniqueness at every intermediate configuration.

Two abstractions account for differences in the mechanised statements. First, Rocq conservatively retains every allocated object name, including private names, and expresses U as `NoDup` of this list. This is stronger than uniqueness of exported definitions alone. Second, the models share an abstract outcome type: rejection, normal return, or an exception carrying an abstract value. Outcome correspondence therefore becomes equality of the abstract outcome lists; native exception representations are not modelled. Linking progress is proved for the abstract name model, while native layout and relocation remain outside the mechanisation.

The development uses only the standard library and has been checked with Rocq 9.2, without admitted proofs or global axioms; the compiler properties remain explicit theorem parameters. Discharging them for MLKit, including joint opening of code and compilation-basis contributions, representation contracts, and native relocation, remains future work.

5 IMPLEMENTATION

The implementation connects MLKit's compilation pipeline to the persistent execution and inspection of native values. We first explain which compiler information must survive an interaction and how the compiler and evaluation processes cooperate. We then describe how serialised bases support later declarations and library loading, how exceptional completion affects the session, and how type descriptions guide the printing of native values.

5.1 Compiler Phases and Retained Information

Each interactive declaration passes through MLKit's compilation pipeline. After parsing, elaboration establishes its static meaning and produces the typed declaration described in Section 3; translation then produces a typed Lambda intermediate language, representing functions, applications, and data operations. Normalisation and Lambda optimisation simplify this representation before memory management is made explicit. Optimisations include inlining and compositional deep argument flattening [10], which can replace nested tuple arguments by their components and pass floating-point components unboxed. The latter records argument transformers describing how later call sites must adapt to a function's chosen calling convention. Retaining these transformers permits optimised calls across compilation units, including successive REPL interactions.

The next phases determine memory placement and lifetime. Region and effect inference assign allocations to regions and introduce their scopes; a region groups values whose storage can be reclaimed together. Multiplicity inference distinguishes regions requiring dynamically growing storage from regions with bounded allocation. After K-normalisation, which names intermediate

computations, storage-mode analysis determines opportunities for storage reuse. Region elimination removes unnecessary region machinery, and physical-size inference computes storage requirements for finite regions. These analyses turn inferred lifetimes into concrete allocation decisions; their development and interaction are surveyed by Tofte et al. [24].

Closure conversion makes function environments explicit. The backend then lowers the program to a statement representation, allocates registers, inserts required loads and stores, computes stack offsets, and performs further simplification before machine-code generation. This organisation builds on the region-aware optimising backend of Elsman and Hallenberg [11]. Assembly and linking produce the shared library loaded by the evaluation process.

The compilation basis collects the environments needed by these phases: source-to-intermediate bindings, optimisation information, region and multiplicity information, and backend representations of exported values. Each phase consumes its part of the accumulated basis and returns information for the new unit, as in Section 3. Thus subsequent declarations can use both the values and the compilation decisions established by earlier ones. The REPL applies the existing pipeline at declaration granularity and coordinates its export information with incremental loading.

Not every declaration generates runtime code. MLKit uses static interpretation of modules [5]: a functor declaration retains its body and the environment needed for subsequent applications at compile time. Thus, in

$$C \vdash d \Rightarrow \exists N.(C', o)$$

o may be the empty object ϵ, for example when d is a functor declaration. All information needed to retain the declaration d is then stored in the compilation basis contribution C', including its references to the preceding basis. A later functor application uses this information to generate code. The empty object requires no runtime work, but the successful interaction still extends the compilation basis to $C + C'$.

5.2 Compilation and Evaluation Processes

The compiler process maintains a list of dependencies identifying the compilation units visible to the next interaction, together with a list of libraries available for linking. After parsing and type-checking a declaration, it reconstructs the required compilation basis, invokes the compiler, and links the generated code into a shared library. It then sends a `LOADRUN` command containing the library's path to the evaluation process.

The evaluation process loads the library using `dlopen`, resolves its initialisation function using `dlsym`, and calls that function. Loading makes the library's exported symbols available to subsequently loaded code. The process persists across interactions, so later declarations can refer to values produced by earlier ones.

After execution, a `PRINT` command supplies a type description and a symbol identifying a value. The evaluation process retrieves and formats the value, then returns the resulting string. The compiler process incorporates this string into its report of the declaration. Values themselves remain in the evaluation process; the protocol transmits descriptions, symbol names, and formatted results.

5.3 Serialisation and Deserialisation

Bases and export bases are serialised and written to disk in so-called basis-files using type-specialised serialisation with sharing [6]. The combinator library builds serializers and deserializers for the basis types, with sharing reducing duplication in stored and restored compiler information. Whenever a toplevel declaration is compiled, bases for previous toplevel declarations are deserialised and assembled to form the basis in which to compile the new toplevel declaration. For performance

reasons, bases are divided into elaboration bases and compilation bases and only the elaboration bases are assembled initially. Then a separate analysis determines which of the compilation bases are needed for the compilation.

5.4 Loading MLB-Files and the Initial Basis

When the REPL is launched, the Standard ML Basis Library [12] is made available to the user through the initial basis. The way this is done is by applying a more general mechanism that allows a user to load any MLB-file. If the MLB-file has been compiled already and its compilation assumptions remain valid, MLKit's recompilation system [7] can reuse the compiled units. Instead, the program unit names that are defined by the basis are added to the current scope. The *current scope* is simply a list of file names identifying bases to be loaded in order to establish the basis in which the program unit is compiled.

When extending the current scope, an optimisation is applied, which builds on the property that for any bases E and F, we have the property that $E + F + E = F + E$. This property allows for pruning the current scope, which reduces the need for deserialisation, in particular if an MLB-file is loaded multiple times, either directly by the user or indirectly through loading of other MLB-files.

5.5 Uncaught Exceptions

The evaluation process implements rule (3) by catching an uncaught exception and returning an `EXN` reply. The compiler process then retains its previous compilation dependencies. A `DONE` reply permits the new export information to enter the context of subsequent interactions. As required by the design in Section 3.5, both cases retain the loaded object. For example, in

```
val r = ref 1;
val a = (r := 0; 5 div (!r));
```

the second interaction reports an exception and introduces no binding for `a`, while the contents of `r` remain zero.

5.6 Type-Indexed Pretty Printing

The evaluation process contains a generic printer, implemented in Standard ML and exported as `pretty_exported`. A `PRINT` request supplies a serialised type description and the symbol of a stored value. The runtime resolves the symbol and passes its value to the printer, which parses the description and interprets it. Thus the compiler supplies the information needed to inspect a value without transmitting the value itself or generating a new printer for each interaction.

A type-indexed interpreter. Consider the following fragment of the type language:

$$\tau ::= \mathsf{int} \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \mid \tau_1 \to \tau_2$$

Record labels are distinct and listed in the compiler's field order. Let w denote a runtime value representation, possibly a pointer into the evaluation process's heap h. Define $\mathit{integer}(w)$ to decode a runtime integer and return its decimal string representation, and $\mathit{field}_i(h, w)$ to retrieve the representation of the ith record field using the runtime's record layout. These operations require that w represents a value of the supplied type. Write $+\!\!+$ for string concatenation and *join* for concatenating a sequence of strings with a separator. The printer $P_h[\tau](w)$ is defined by interpretation of the

type description:

$$P_h[\mathsf{int}](w) = \mathit{integer}(w)$$
$$P_h[\{l_1 : \tau_1, \ldots, l_n : \tau_n\}](w) = \texttt{"\{"} \mathbin{+\!\!+} \mathit{join}\big(\texttt{","}, [\, l_i \mathbin{+\!\!+} \texttt{"="} \mathbin{+\!\!+} P_h[\tau_i](\mathit{field}_i(h, w)) \,]_{i=1}^{n}\big) \mathbin{+\!\!+} \texttt{"\}"}$$
$$P_h[\tau_1 \to \tau_2](w) = \texttt{"fn"}$$

The empty record is printed as `"{}"` in this presentation; the implementation prints unit using tuple syntax `"()"`. The function case neither calls the function nor traverses its closure. For example, given the declaration

```
val r = {count = 7, step = fn x ⇒ x + 1};
```

the description $\{\mathit{count} : \mathsf{int}, \mathit{step} : \mathsf{int} \to \mathsf{int}\}$ directs the printer to retrieve two fields and produce `"{count=7,step=fn}"`.

Untagged tuples and records. The type description provides field labels, field types, and arity; these need not be stored with each record value. The implementation's `selectTuple` primitive retrieves a field through the runtime's record-access operation, accounting for its layout. Tuples use the same traversal with positional types and different punctuation. In particular, an untagged record need not identify its own shape: the type drives recursive inspection. The printer relies on agreement between the supplied description and the compiled representation; its low-level projections and casts do not independently check that agreement.

Unboxed data types. For algebraic data types, the description additionally carries type names, constructor schemes, and the compiler's representation choice. The printer identifies the constructor, recovers its source name, instantiates its argument type with the supplied type arguments, and recursively prints the payload. Enumerations use immediate constructor codes, while an untagged single-constructor wrapper passes its value directly to the payload printer. Boxed constructors require projection from an allocated constructor block.

MLKit also uses double-ended bit-stealing [8], which encodes constructor information in available low or high bits of a value word, avoiding a separate constructor allocation for eligible data types. The printer follows the recorded representation choice: for a low-bit encoding it extracts the constructor code and removes its low tag bits before inspecting a payload; for a high-bit encoding it extracts and clears the high tag. The latter preserves any low-bit encoding belonging to the payload, which its own type description subsequently interprets. Printing therefore follows the same nested representation choices as compiled pattern matching.

The full interpreter also handles primitive types, lists, options, and references. The `pretty_depth` setting bounds recursive constructor and reference inspection, and `pretty_string_size` limits displayed string prefixes. These controls keep recursive structures and long strings manageable in an interactive session.

6 EVALUATION OF THE APPROACH

We evaluate functional behaviour with regression tests and isolate the cost of incremental dynamic linking with a loader microbenchmark. The latter tests whether retaining many loaded objects makes subsequent loads more expensive; it does not measure the complete latency of an interactive declaration.

6.1 Functional Tests

The existing REPL suite passes all eight configurations: six scripts with the Basis Library and two basic scripts without it. These cover arithmetic, library loading, Basis operations, changes of

working directory, and printing of nested records and several datatype representations. Additional interactions check that a closure retains its original binding after source-level shadowing, that an exception preserves prior reference updates but introduces no binding for the failed declaration, and that the session continues after the resulting unbound-identifier error. Separate runs in MLKit and ReML both preserve the reference update and continue after `Div`.

These checks use the installed x86-64 MLKit and ReML binaries, identifying themselves as version `4.7.22-33-g906d563f-dirty`. They provide regression evidence for the exercised cases; they do not discharge the compiler properties of Section 4.

6.2 Incremental Linking Cost

The experiment runs on an arm64 Mac with macOS 26.5.1, using x86-64 code under Rosetta to match the installed compiler's target. Apple Clang 21.0.0 generates 1,024 distinct shared libraries before timing. Each library exports a value and an initialiser; except for the first, it imports the preceding library's value. The initialiser increments that value, and the driver checks the result at every step. All loaded libraries remain resident. Each timed step comprises `dlopen` with `RTLD_NOW|RTLD_GLOBAL`, `dlsym` of the new initialiser, and its execution, matching the operations in Section 5.

One complete warm-up precedes five measured runs, each in a fresh process. The measurements exclude library generation, process startup, and output formatting. In particular, warming the generated code excludes first-use translation costs under Rosetta. The accompanying `evaluation/` directory contains the driver, individual timings, and test transcripts.

We repeat the experiment on a ThinkPad with an Intel Core i7-1360P, running Ubuntu 26.04.1, Linux 7.0.0-31, and glibc 2.43. GCC 15.2.0 builds native x86-64 shared objects with `-O2 -fPIC -shared`. The generated dependency chain, loading flags, warm-up, correctness checks, and five-run protocol are unchanged; the compiler, object format, loader, and hardware differ. All loads and value checks also succeed on this platform.

Table 1 reports medians across the five runs on each platform. Each row uses the corresponding prefix of the 1,024-load trace; the per-load columns average the final 128 operations in that prefix before taking the median across runs. All initialisers returned the expected value in every run.

We also rebuilt the macOS libraries and inspected their Mach-O sections and symbol tables with `size` and `nm`. The code and file columns of Table 1 distinguish instruction bytes in the `__text` sections from complete library files, including headers, linking metadata, and alignment. Sizes and name counts are accumulated over the loaded prefix; KiB and MiB denote 2^{10} and 2^{20} bytes, respectively.

Table 1. Incremental loading after warm-up, with accumulated macOS sizes and generated name counts

	macOS / Rosetta				Linux / native		Names	
Libraries	Total (s)	ms/load	Code (KiB)	Files (MiB)	Total (s)	ms/load	Exports	Imports
128	0.197	1.54	3.62	1.015	0.0026	0.0205	256	127
256	0.724	4.12	7.25	2.034	0.0058	0.0257	512	255
512	2.930	9.85	14.50	4.073	0.0157	0.0464	1,024	511
1,024	12.264	22.81	29.00	8.152	0.0504	0.0832	2,048	1,023

Each library defines two distinct names: its value and its initialiser. After N loads there are therefore $2N$ generated exported names and $N - 1$ distinct imported names. The imports refer to earlier exports, so they introduce no additional names. These counts exclude symbols provided by the driver and system libraries. At 1,024 libraries, the 29,692 instruction bytes occupy files totalling

8,547,880 bytes; the sum of their Mach-O segment sizes is 16,773,120 bytes, approximately 16 MiB. Segment sizes describe virtual address-space requirements, not resident memory, and exclude Rosetta translations and additional loader data. Thus this experiment accumulates many small objects and names while its actual instruction volume remains modest.

The macOS results demonstrate successful incremental loading at this scale, but do not support constant cost per interaction. Doubling the retained collection from 512 to 1,024 libraries increases cumulative time by about a factor of four; the observed cumulative trend is roughly quadratic over this range. The five full runs range from 12.01 to 12.70 seconds. The experiment does not separate symbol lookup from other loader bookkeeping, so it does not identify the cause of this growth or establish an asymptotic bound.

On the ThinkPad, the median total for 1,024 loads is 0.0504 seconds, with full runs ranging from 0.0500 to 0.0595 seconds, about 243 times faster than the macOS/Rosetta median. Nevertheless, the mean cost of the final 128 loads grows from 0.0205 ms at 128 libraries to 0.0832 ms at 1,024 libraries. Thus increasing per-load cost also occurs without Rosetta, although its magnitude is much smaller here. The comparison does not isolate Rosetta's contribution: native arm64 measurements on the same Mac would be needed to separate it from hardware and loader differences. The Linux shared-object files total 15,532,120 bytes (14.81 MiB); the generated name counts are unchanged.

These results are specific to the tested platforms, dependency pattern, and warm-cache protocol. A complete scalability evaluation must also measure compilation, basis deserialisation, printing, and memory use in long REPL sessions. In particular, retained libraries occupy memory, and garbage collection is disabled in the tested MLKit REPL. The name-uniqueness invariant ensures that linking does not redefine an existing symbol; it provides no complexity or space bound.

7 RELATED WORK

Interactive ML systems already combine compilation, persistent bindings, and value inspection. SML/NJ's interactive system accepts declarations and source files and provides controls for printing structured values [22]. Appel [1] describes how SML/NJ makes the compiler available through a Standard ML structure, allowing ML programs to invoke compiler facilities. Poly/ML exposes compilation through `PolyML.compiler`, which compiles a toplevel phrase into a `unit -> unit` function; executing it performs effects and adds bindings to the namespace. Its `addPrettyPrinter` facility associates printers with types [18]. Our design makes the connection to MLKit's existing compilation pipeline explicit through retained compilation bases, shared-library loading, and runtime interpretation of serialised type descriptions.

OCaml's traditional toplevel executes bytecode, while its experimental `ocamlnat` toplevel supports native execution. The `#install_printer` directive selects user printers by matching their argument types to the types of displayed values [17]. Thus native interactive execution and type-directed printer selection both have precedents. Here the printing problem is specifically to interpret MLKit's untagged records and unboxed constructors using representation information supplied by the compiler.

GHCi supports both bytecode and compiled object code, and its external interpreter option executes code in a separate process communicating with GHC over pipes [13]. This is closely related to our compiler/evaluator separation. Ordinary GHCi expression printing uses `print` and a `Show` instance; our generic printer instead interprets a type description to inspect a native value. GHCi also provides separate debugger commands for inspecting values without requiring their full evaluation.

Cling provides a non-ML example of interactive execution built on an existing compiler. It retains Clang state across inputs, lowers new fragments to LLVM IR, and uses LLVM's JIT to execute them [25]. Both systems reuse a compiler pipeline and retain information across interactions; our

design exposes that information as compilation bases and executes shared libraries in a separate evaluation process.

Verified interactive compilation is central to CakeML. Kumar et al. [15] present an ML REPL implemented in x86-64 machine code and verified in HOL4, including incremental compilation and correctness of its printed results. Sewell et al. [20] extend the modern CakeML compiler with `Eval`, enabling runtime compilation whose generated code shares higher-order values and makes recursive calls with existing code. These works verify concrete implementations of interactive or dynamic execution. Our formal development identifies the compiler properties needed for session simulation; establishing those properties for MLKit remains separate work.

Smart recompilation aims to avoid rebuilding units when changes to their dependencies do not invalidate the assumptions used to compile them. Shao and Appel's *smartest recompilation* [21] infers minimal import interfaces for ML compilation units. Elsman's framework [7] addresses cut-off incremental recompilation in the presence of inter-module optimisation. We use its compilation bases to compile successive declarations, but recompilation management is outside the scope of this paper: we do not consider change detection, invalidation, or scheduling the rebuilding of previously compiled units. Library loading delegates such decisions to MLKit's existing manager.

Cardelli's linksets [2] provide the interface and substitution structure used in Section 3.4; our adaptation adds fresh instantiation of object names and execution of each new initialisation against a retained loaded collection. Glew and Morrisett [14] extend this line of work to typed assembly object files, supporting abstract types and higher-order type constructors and proving that linking preserves type safety. Their result concerns the meaning of object interfaces as well as their compatibility. Our model isolates symbol uniqueness and resolution, while correctness of representation contracts and native relocation remains an implementation obligation.

Compositional CompCert [23] verifies separate compilation in Coq using linking semantics and simulations that account for shared-memory interaction between modules. It illustrates the additional reasoning needed to establish compiler properties across module boundaries. Our session induction composes successive interactions under assumed declaration-level properties; it does not verify the compiler passes or their shared-memory representations.

8 CONCLUSION AND FUTURE WORK

Incremental compilation provides a foundation for interactive native execution in MLKit and ReML. Retained compilation bases carry the information needed by subsequent declarations, while incremental linking connects their generated code to a persistent runtime. Type-indexed pretty-printing makes native values available for inspection. The model separates source bindings from generated symbols, and the Rocq development shows, under explicit compiler properties, how declaration-level correctness yields finite-session simulation without duplicate linked names.

Regression tests exercise the implementation, and the loader experiment shows that accumulated linking costs depend substantially on the platform. Future work includes measuring complete interaction latency and discharging the compiler and loader obligations for MLKit, connecting the mechanised model to its representations and compilation phases.

ACKNOWLEDGEMENT

The development was assisted using GPT-6 Astra.

REFERENCES

[1] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

[2] Luca Cardelli. 1997. Program Fragments, Linking, and Modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, NY, USA, 266–277. https://www.microsoft.

com/en-us/research/publication/program-fragments-linking-and-modularization/

[3] Martin Elsman. 1998. Polymorphic Equality - No Tags Required. In *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*. Springer-Verlag, Berlin, Heidelberg, 136–155.

[4] Martin Elsman. 1999. *Program Modules, Separate Compilation, and Intermodule Optimisation.* Ph. D. Dissertation. Department of Computer Science, University of Copenhagen. https://elsman.com/pdf/phd.pdf

[5] Martin Elsman. 1999. Static Interpretation of Modules. In *Procedings of Fourth International Conference on Functional Programming (ICFP'99)*. ACM Press, 208–219.

[6] Martin Elsman. 2005. Type-Specialized Serialization with Sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*. https://elsman.com/pdf/TFP05final_mael.pdf

[7] Martin Elsman. 2008. *A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization.* Technical Report. IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark.

[8] Martin Elsman. 2024. Double-Ended Bit-Stealing for Algebraic Data Types. *Proceedings of the ACM on Programming Languages* 8, ICFP, Article 239 (Aug. 2024), 33 pages. https://doi.org/10.1145/3674628

[9] Martin Elsman. 2024. Explicit Effects and Effect Constraints in ReML. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 79 (Jan. 2024), 25 pages. https://doi.org/10.1145/3632921

[10] Martin Elsman. 2025. Compositional Deep Argument Flattening. In *Informal Proceedings of the ML Family Workshop*. Singapore. https://elsman.com/pdf/ml25-deep-flattening.pdf

[11] Martin Elsman and Niels Hallenberg. 1995. An Optimizing Backend for the ML Kit Using a Stack of Regions. Student Project 95-7-8, University of Copenhagen (DIKU).

[12] E. R. Gansner and J. H. Reppy (Eds.). 2004. *The Standard ML Basis Library.* Cambridge University Press.

[13] GHC [n. d.]. *GHC User's Guide: Using GHCi.* GHC. https://downloads.haskell.org/ghc/9.12.2/docs/users_guide/ghci.html Accessed September 2026.

[14] Neal Glew and Greg Morrisett. 1999. Type-Safe Linking and Modular Assembly Language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 250–261. https://glew.org/nglew/papers/mtal.pdf

[15] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 179–191. https://doi.org/10.1145/2535838.2535841

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised).* MIT Press.

[17] OCaml [n. d.]. *The OCaml Manual: The Toplevel System or REPL.* OCaml. https://ocaml.org/manual/5.4/toplevel.html Accessed September 2026.

[18] Poly/ML [n. d.]. *The PolyML Structure.* Poly/ML. https://polyml.org/documentation/Reference/PolyMLStructure.html Accessed September 2026.

[19] Claudio V. Russo. 1999. Non-dependent Types for Standard ML Modules. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP '99)*. Springer-Verlag, 80––97. https://doi.org/10.5555/645815.668885

[20] Thomas Sewell, Magnus O. Myreen, Yong Kiam Tan, Ramana Kumar, Alexander Mihajlovic, Oskar Abrahamsson, and Scott Owens. 2023. Cakes That Bake Cakes: Dynamic Computation in CakeML. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 152 (2023), 24 pages. https://doi.org/10.1145/3591266

[21] Zhong Shao and Andrew W. Appel. 1993. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 439–450. https://flint.cs.yale.edu/flint/publications/sepcomp.pdf

[22] SML/NJ [n. d.]. *Standard ML of New Jersey Interactive System.* SML/NJ. https://smlnj.org/doc/interact.html Accessed September 2026.

[23] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 275–287. https://doi.org/10.1145/2676726.2676985

[24] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (01 Sep 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[25] Vassil Vassilev. 2020. Interactive C++ with Cling. The LLVM Project Blog. https://blog.llvm.org/posts/2020-11-30-interactive-cpp-with-cling/