A Canonical Routine Form for Compositional Computation

Abstract.
This paper proposes a formal model of compositional computation based on the factorization of computational processes and a rigorous description of their structural properties. The model is defined by a set of definitions and rules that represent computations as a canonical composition of elementary operations with explicitly specified dependencies. It is shown that such factorization yields a linearly ordered representation of computations, simplifies the analysis of their structure, and enables the identification of invariant properties of the compositional form that remain implicit in traditional descriptions of computational processes. The proposed approach provides a foundation for a unified representation of compositional computations and may serve as a basis for further research on formal models in theoretical computer science.

Introduction:
A central motivation for introducing a canonical routine form is to make the structural content of compositional computations explicit. Traditional models often describe computation either through high-level functional composition or through low-level operational semantics, leaving the relationship between structure, dependencies, and execution order implicit. The proposed formulation bridges this gap by providing a representation in which compositional structure, execution order, and data dependencies are simultaneously explicit and formally analyzable.

1. Problem Statement

Let a regularity be given by a mapping

$$\Phi(x_0, x_1, \dots, x_N) = y,$$

where $x_i \in X$ are arguments, $y \in X$ is the result, and X denotes the set of admissible values.
The objective is to describe a structured method for computing the value y by means of simpler functions and to formalize the structure of a computational process that implements such a computation.
It is assumed that the considered regularity admits a compositional representation, that is, it can be expressed through a sequential application of functions of lower arity.

2. Factorization of Compositional Computation

This section introduces a stepwise factorization of a compositional computation, leading to a canonical compositional representation that will serve as the basis for subsequent constructions.

2.1. Sequential compositional representation

Assume that there exists a sequence of binary functions

$$f_1, f_2, \dots, f_N,$$

such that the intermediate values are defined recursively by

$$y_0 := x_0, \quad y_n := f_n(y_{n-1}, x_n), \qquad n = 1, \dots, N,$$

and the final value is given by

$$y := y_N.$$

Then the equality

$$y = \Phi(x_0, x_1, \dots, x_N)$$

holds.
This representation expresses the original regularity Φ as a sequential application of binary functions.

2.2. Factorization by repeating functions
In the general case, the functions f_n may coincide. To account for this, introduce a family of distinct binary functions

$$\{g_k\}_{k \in K}, \qquad g_k : X \times X \to X,$$

together with a mapping

$$k : \{1, \dots, N\} \to K,$$

such that

$$f_n = g_{k(n)} \text{ for all } n.$$

Under this assumption, the recursive computation takes the form

$$\begin{aligned} y_1 &= g_{k(1)}(x_0, x_1), \\ y_2 &= g_{k(2)}(y_1, x_2), \\ &\vdots \\ y &= g_{k(N)}(y_{N-1}, x_N), \end{aligned}$$

and can equivalently be written as the nested expression

$$y = g_{k(N)}(g_{k(N-1)}(\dots g_{k(1)}(x_0, x_1), x_2 \dots), x_N)).$$

This step reduces the number of distinct functions involved in the computation and yields a factorized representation with respect to repeated function usage.

2.3. Unary transformations and canonical compositional form
In the recursive scheme

$$y_0 := x_0, \quad y_n := g_{k(n)}(y_{n-1}, x_n), \quad n = 1, \dots, N,$$

the second argument x_n is fixed at each step. Consequently, each step can be regarded as the application of a unary transformation to the current intermediate value.
For any $k \in K$ and $x \in X$, define the unary mapping

$$f_{k,x}: X \to X,\ f_{k,x}(u) := g_k(u, x).$$

In these terms, the recursive computation becomes

$$y_0 := x_0, \quad y_n = f_{k(n),x_n}(y_{n-1}), \qquad n = 1, \dots, N,$$

with the final value

$$y := y_N.$$

Thus, the computation can be expressed as a composition of unary transformations applied to the initial value:

$$y = f_{k(N),x_N} \circ f_{k(N-1),x_{N-1}} \circ \dots \circ f_{k(1),x_1}(x_0).$$

For compact notation, introduce the composition operator

$$\bigcirc_{n=1}^{N} f_{k(n),x_n}(x_0) := f_{k(N),x_N} \circ f_{k(N-1),x_{N-1}} \circ \dots \circ f_{k(1),x_1}(x_0).$$

The computation then takes the canonical compositional form

$$\boxed{\bigcirc_{n=1}^{N} f_{k(n),x_n}(x_0) = y}. \quad (1)$$

For notational convenience, introduce an index z_n uniquely identifying the pair $(k(n), x_n)$ and define

$$f_{z_n} := f_{k(n),x_n}$$

In this notation, expression (1) can be rewritten as

$$\bigcirc_{n=1}^{N} f_{z_n}(x_0) = y,$$

which is a purely notational abbreviation and introduces no additional computational assumptions.

3. Parametric Generalization of the Compositional Form

This section considers a parametric generalization of the compositional form of computation to the case of a family of independent tasks. The compositional structure of the computation is preserved, and no additional computational assumptions are introduced.

3.1. Family of independent tasks

Let a family of tasks indexed by a parameter

$$m = 1,2, \dots, M$$

be given.
Each task mmm is characterized by:

- an initial value $x_{m,0} \in X$;
- a sequence of fixed arguments $\{x_{m,n}\}_{n=1}^{N_m}$;
- the length of the compositional chain N_m;
- a computed result $y_m \in X$.

It is assumed that each task admits a representation in the form of a compositional computation whose structure is consistent for all values of the parameter m.

3.2. Compositional form of an individual task

For a fixed value of the parameter mmm, the computation of the task result can be written as

$$\bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}(x_{m,0}) = y_m.$$

Here:

- $k_m(n)$ specifies the selection of a function from the admissible family;
- the second argument $x_{m,n}$ is fixed at each step of the composition;
- each mapping $f_{k,x}: X \to X$ is defined via a binary function

$$f_{k,x}(u) = g_k(u, x),$$

where $g_k: X \times X \to X$.

3.3. Joint representation of a family of tasks

Since all tasks employ the same compositional computation mechanism, the expression above can be written simultaneously for all values of the parameter m:

$$\bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}(x_{m,0}) = y_m, \quad \forall m = 1, \dots, M.$$

This representation defines a parametric generalization of the compositional form, in which the task parameter mmm is external to the compositional scheme itself.

3.4. Remark on the nature of the generalization

The parametric representation considered above:
does not modify the structure of the compositional computation;
does not introduce new elementary transformations;
differs solely by the parametrization of initial values, arguments, and the length of the compositional chain.

3.5. Concluding remark

This section establishes a parametric representation of a family of compositional computations without addressing issues of global state, execution order, or routine organization of computation. These aspects are considered in the subsequent sections.

4. Sequential Execution of a Family of Tasks with a Shared State

This section considers an organization of the computational process in which a family of tasks is executed sequentially on a single computational device using a shared set of values. The compositional structure of the computation of each individual task is preserved.

4.1. Shared set of values

Let the computational device operate on a shared set of values

$$\{x_i\}_{i\in I} \subseteq \mathrm{X},$$

which is used to store input values, intermediate results, and output data.

For each task $m = 1,2, \dots, M$, the following indices are fixed:

- $I_m \in I$, specifying the element of the shared set used as the input value of the task;
- $P_m \in I$, specifying the element of the shared set into which the result of the task is written.

The results of individual tasks do not form a distinguished set and are treated as elements of the shared set of values.

4.2. Compositional form of a task under a shared state

For each task m a finite sequence of unary transformations is given,

$$\{f_{k_m(n),x_{m,n}} : X \to X\}_{n=1}^{N_m},$$

where each transformation is determined by a fixed choice of a function and a fixed value of the second argument.

With respect to the introduced shared set of values, the computation of the result of task m is described by the compositional expression

$$\bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}\left(x_{I_m}\right).$$

This expression defines the value that is subsequently treated as the result of the task and is to be written into the shared state of the computational device.

4.3. Routine sequence of task execution

Define the operator

$$R_{m=1}^{M}[\cdot],$$

which denotes a sequential routine of task executions indexed by

$$m = 1,2, \dots, M.,$$

performed in a strict order, without parallelism.

For each value of m, the execution of the routine includes the following steps:

- the input value x_{I_m} is selected from the shared set of values;
- the result of the task is computed as

$$y_m := \bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}(x_{I_m}).$$

- the obtained value is written into the corresponding element of the shared state,
-

$$x_{P_m} := y_m.$$

The assignment at step m may modify values that are subsequently used during later steps of the routine.

4.4. Formal representation of sequential execution

The procedure of sequential task execution can be compactly written as

$$R_{m=1}^{M}[x_{P_m} := \bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}(x_{I_m})].$$

The symbol "$:=$" is interpreted as an assignment operation reflecting a modification of the shared state of the computational device.

4.5. Final state of computation

Since the tasks are executed strictly sequentially, the state of the set $\{x_i\}_{i\in I}$ after completion of step $m = M$ constitutes the final state of the computational process.
The final result of the computation is defined as the value written into the element of the shared state corresponding to the last executed assignment.

4.6. Concluding remark

Thus, this section establishes a transitional form of the computational process in which a family of tasks is executed sequentially while preserving the compositional structure of each task and using a shared state. This representation serves as the basis for reducing the computation to the canonical routine form considered in the subsequent sections.

5. Reduction of the Compositional Form to the Canonical Routine Form

In this section, the computational process is reduced to a canonical form in which it is fully described by a sequential routine of elementary steps operating on a shared state. This form is intended to represent and solve a family of tasks within a single computational process, taking into account the coincidence of used functions and arguments.

5.1. Initial transitional form

As a starting point, consider the transitional form of the computational process that specifies the sequential execution of a family of tasks:

$$R_{m=1}^{M}[x_{P_m} := \bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}(x_{I_m})].$$

This notation describes a two-level linear structure of computation: an outer ordering over the task index m, defined by the routine, and an inner ordering over the index n, defined by the compositional chain of each task.

5.2. Total number of elementary steps
Each task m contains exactly N_m elementary applications of unary transformations. Since the tasks are executed strictly sequentially, the total number of elementary steps in the entire computational process is

$$Q := \sum_{m=1}^{M} N_m.$$

The value Q determines the total length of the computation when all tasks are executed sequentially on a single computational device.

5.3. Unfolding of the compositional structure
Each composition

$$\bigcirc_{n=1}^{N_m} f_{k_m(n),x_{m,n}}$$

represents a finite sequence of elementary transformation applications. Consequently, the entire collection of computations defined by the transitional form can be unfolded into a single linearly ordered sequence of Q elementary steps acting on the shared state.
Under this unfolding, the parameters m and n are used exclusively to define the execution order of steps and are not required for the further description of the computational process.

5.4. Canonical routine form
Let the computational device operate on a shared set of values

$$\{x_i\}_{i\in I} \subseteq X,$$

and let a family of binary functions

$$g_k : X \times X \to X$$

be given.
Each elementary computational step is determined by indices

$$T(i), S(i), L(i) \in I, \qquad K(i) \in K,$$

and has the form of an assignment operation

$$x_{T(i)} := g_{K(i)}\left(x_{S(i)}, x_{L(i)}\right), \qquad i = 1,2,\dots,Q.$$

The complete computational process is specified by the canonical routine form

$$R_{i=1}^{Q}[x_{T(i)} := g_{K(i)}(x_{S(i)}, x_{L(i)})], \qquad i = 1,2,\dots,Q. \tag{2}$$

5.5. Coincidence of functions and arguments
Within this canonical form:

- all functions g_k are taken from a single family and are not duplicated;
- identical arguments x_i may be used in multiple elementary steps without copying;

- coincidence of functions and arguments is naturally accounted for through repeated use of the indices $K(i), S(i)$, and $L(i)$.

Thus, a family of tasks with a shared computational structure is reduced to a single routine in which redundant repetition of both functions and argument values is eliminated.

5.6. Nested regularities as a special case

Tasks containing nested regularities can be represented within this canonical form as a collection of individual tasks connected through the transfer of argument values via the shared set of values $\{x_i\}$. In this case:

- each nested regularity is represented as an independent sequence of elementary steps;
- the results of such steps are written into elements of the shared state;
- the obtained values may subsequently be used as arguments by other tasks.

Data transfer between tasks is performed directly through the shared state, without introducing additional control mechanisms.

5.7. Final value of computation

Since the execution of elementary steps is strictly sequential, the final state of the computation is determined by the values obtained after the execution of the last step. The final result of the computational process is defined as

$$y := x_{T(Q)}.$$

Thus, a computational process originally specified in compositional form can be reduced to a canonical form of a sequential routine operating on a shared state. In this form, multiple tasks, coinciding functions and arguments, and nested regularities are all reduced to a single linearly ordered sequence of elementary binary assignment operations.

6. Partial Modification of Functions in the Canonical Routine Form

Consider the canonical routine form of computation defined by a sequential routine of elementary steps operating on a shared state. Introduce an additional binary control sequence

$$S(i) \in \{0,1\}, \qquad i = 1,2,\dots,Q,$$

which determines the type of the elementary step of the computational process.
For each step i, two possible actions are distinguished.

6.1. Function application

When $S(i) = 0$, a standard computational step is performed: the value of the function is computed using the current argument values, and the result is written into the corresponding element of the shared state:

$$x_{T(i)} := g_{K(i)}(x_{S(i)}, x_{L(i)}).$$

This step fully corresponds to the elementary transformation used in the canonical routine form.

6.2. Partial modification of a function

When $S(i) = 1$, the elementary step is used to modify a function rather than to compute its value. In this case, the mapping $g_{K(i)}$ is redefined at a specific argument pair without changing the set of arguments:

$$g_{K(i)}(x_{S(i)}, x_{L(i)}) := x_{T(i)}.$$

It is assumed that the modification affects only the value of the function at the point $(x_{S(i)}, x_{L(i)})$, while the function remains unchanged on all other arguments.
Thus, the canonical routine form admits both the computation of function values and their partial modification during the execution of the computational sequence.

6.3. Generalized routine form

Here, $f_i^{S(i)}$ is not an independent function symbol, but a notational abstraction denoting one of the two admissible actions applied to the underlying binary function $g_{K(i)}$ at step i.
Taking the control sequence $S(i)$ into account, the canonical routine form can be written in the following generalized form:

$$\boxed{R_{i=1}^{Q}\left[f_i^{S(i)}\left(x_{S(i)}, x_{L(i)}\right) \to x_{T(i)}\right]}. \qquad (3)$$

The arrow "→" denotes an operational effect rather than a logical implication.

- f_i^0: "apply" — calculate and assign $x_{T(i)} := g_{K(i)}(x_{S(i)}, x_{L(i)})$
- f_i^1: "update" — locally override $g_{K(i)}(x_{S(i)}, x_{L(i)}) := x_{T(i)}$.

The notation $g(x, y) := v$ denotes a pointwise update of the function value at the argument pair (x, y), leaving all other values unchanged.

Here, the notation $f_i^{S(i)}$ indicates the selection of one of the two actions defined above, depending on the value of the control variable $S(i)$, and does not represent logical implication or equivalence.
The control variable $S(i)$ does not affect the direction of assignment: the target index $T(i)$ is fixed for each step. Instead, $S(i)$ selects whether the step updates a state value or modifies a function at a specific argument pair.

6.4. Remark on the nature of the generalization

The generalized routine form:

- does not alter the basic structure of the computational process;
- preserves a unified shared state of values and functions;
- allows the computational scheme to change during execution through local function modification.

As a result, the computational process acquires the ability to adapt its behavior without introducing additional control constructs or changing the execution order of elementary steps.

6.5. Concluding remark

The introduction of partial function modification extends the canonical routine form by allowing the description of computational processes in which functions may change during execution. At the same time, the computation remains a linear sequence of elementary steps over a shared state, and all permissible operations are fully determined by the control sequence $S(i)$.

7. Control Flow via Argument-Guided Execution

This section introduces a mechanism for non-linear control flow within the proposed computational framework, achieved without introducing external control structures or global branching constructs. Instead, the execution order is determined locally through the modification of designated arguments.

7.1 Execution Index and Computational State

Let the execution of a computational routine be indexed by a discrete step counter

$$i \in \{1,2, \dots, Q\},$$

where Q denotes the total number of elementary computational steps defined in Section 5.
At each step i, the global computational state is represented as

$$\sum_i = (\{x_j\}_{j \in I}, i),$$

where $\{x_j\}$ is the current set of argument values and i is the active execution index.
Each step i is associated with a computational rule $g_{K(i)}$, selected according to the routine structure introduced earlier.

7.2 Default Sequential Semantics

In the absence of explicit control intervention, execution proceeds sequentially.
Given state Σ_i, application of the rule $g_{K(i)}$ produces an updated argument set $\{x'_j\}$, and the next execution index is defined as

$$i_{next} = i + 1,$$

provided that $i < Q$. Execution terminates when $i = Q$.
This defines a purely linear execution order consistent with the canonical routine form introduced in Section 5.

7.3 Argument-Guided Step Selection

To enable non-linear execution, a designated argument x_p is interpreted as a control argument.
We say that x_p is modified at step i if the assignment of step i writes into location p, i.e., $T(i) = p$.
If, during execution of step i, the value of x_p is modified, its updated value is interpreted as a candidate execution index for the next step.
Formally, the next execution index i' is determined as

$$i' = \begin{cases} x_p, if\ x_p \in \{1, \dots, Q\}, \\ i + 1, otherwise. \end{cases}$$

Thus, the execution flow may be redirected to any valid step index through local modification of the argument state, while invalid or out-of-range values default to sequential progression.

7.4 Locality and Determinism

The step-selection mechanism is strictly local:

- the rule applied at step i depends only on the current execution index i,

- the next execution index depends only on the locally updated argument x_p,
- no global control structures or external schedulers are introduced.

For a fixed routine structure and initial argument state, execution remains deterministic.

7.5 Expressive Power of the Control Mechanism

This argument-guided execution mechanism enables the representation of standard control-flow patterns, including:

- conditional branching,
- indirect jumps,
- loops with dynamically determined iteration counts,
- state-dependent execution redirection.

All such behaviors are realized within the same uniform computational form, without extending the underlying rule structure or introducing additional control primitives.

7.6 Relation to the Canonical Form

The control mechanism described above does not alter the canonical linear representation of the routine introduced in Section 5. Instead, it defines an operational interpretation under which the linear sequence of steps may be traversed non-linearly at runtime.

Accordingly, the canonical form serves as a structural description of the computation, while the argument-guided execution semantics defines its dynamic behavior. This control mechanism is orthogonal to the action-selection variable : $S(i)$ introduced in Section 6: : $S(i)$ selects the type of an elementary step, while x_p may affect the next execution index.

7.7 Summary

By interpreting selected arguments as execution indices, non-linear control flow is achieved through strictly local transformations of the computational state. This approach preserves the uniform structure of the computational model while enabling flexible execution patterns within a formally defined operational semantics.

8. Compact Routine Form with a Fixed Functional Basis

This section considers a simplified canonical routine form of computation obtained by fixing the functional basis. It is shown that computational expressiveness of the full routine form is preserved, while the amount of required control description is significantly reduced.

8.1. Fixed functional basis

Assume that the canonical routine form employs a single fixed binary function

$$g: X \times X \to X.$$

In this case, the choice of function at each elementary computational step does not depend on the step index, and the family of functions $\{g_k\}$ degenerates into a single function g. Accordingly, the function-selection index becomes redundant and can be eliminated from the description of the computational process.

8.2. Compact routine form

With a fixed functional basis, the canonical routine form takes the form

$$R_{i=1}^{Q}[x_{T(i)} \coloneqq g(x_{S(i)}, x_{L(i)})].$$

The final result of the computational process, as in the full routine form, is determined by the value of the shared state element after the last step:

$$y \coloneqq x_{T(Q)}.$$

This form is henceforth referred to as the **compact version of the computer formula**.

8.3. Equivalence to the full canonical routine form

Despite fixing the functional basis, the compact routine form retains all computational capabilities of the full canonical routine form introduced in the preceding sections. All distinctions between elementary computational steps are specified solely by the argument and target indices

$$S(i), L(i), T(i).$$

Control of the computational process, including conditional branching, loops, and changes in computational behavior, is implemented through argument values and the shared state, without the need to extend the functional basis.

8.4. Resource efficiency

Eliminating the function-selection index leads to a substantial reduction in the control description of the computational process. In particular:

- the amount of information required to specify each elementary step is reduced;
- the size of the functional basis remains fixed, independent of the complexity of the tasks being solved;
- extension of computational capabilities is achieved through argument structure and step sequencing rather than by increasing the number of functions.

Thus, the compact routine form requires comparatively fewer resources for description and implementation than the full canonical routine form.

8.5. Computational expressiveness of the compact form

The compact version of the computer formula solves all tasks solvable by the full canonical routine form, while preserving the linear structure of elementary steps and a unified shared state.
In this sense, the compact routine form is computationally equivalent to the full canonical routine form, while being more economical in terms of descriptive and control resources.

8.6. Concluding remark

Thus, this section derives compact routine form that possesses the same computational expressiveness as the full routine form, but requires comparatively fewer resources. This makes the compact routine form a convenient foundation for formal analysis, implementation, and further generalizations of computational models.

9. Structural Properties and Advantages of the Canonical Routine Form

This section summarizes the principal structural properties of the proposed computational model and highlights the advantages obtained by representing compositional computations in the canonical routine form.

9.1. Canonical linear ordering of computation

The model transforms any compositional computational structure into a strictly linear sequence of elementary steps. This yields a fully determined execution order, enables precise tracing of intermediate states, and eliminates ambiguities inherent in partially ordered or implicitly defined execution models. Such linearization significantly simplifies formal analysis, transformation, and optimization of computational processes.

9.2. Unified global state as a universal data carrier

All computational dynamics are expressed exclusively through modifications of a single global set of values. This provides a unified representation for multiple tasks, eliminates the need for auxiliary data structures, and yields a memory model closely aligned with the operational principles of real computational devices.

9.3. Canonical routine form as a normal form of computation

The canonical routine representation functions as a normal form for compositional computations: any compositional process can be reduced to this form without loss of information. This enables direct structural comparison between different computations and provides a uniform foundation for formal reasoning, analogous in role to normal forms in other formal systems.

9.4. Natural elimination of redundancy

The representation inherently eliminates redundancy. Repeated use of identical functions is captured through shared function indices, while repeated use of identical arguments is represented through shared value indices. As a result, the computational structure becomes compact, factorized, and free of unnecessary duplication, without introducing additional abstraction mechanisms.

9.5. Support for nested regularities without structural extension

Nested computational regularities require no additional constructs. They are represented as contiguous segments of the same routine sequence, with data exchange performed through the global state. The linear structure of computation remains unchanged regardless of the depth or complexity of nesting.

9.6. Parameterized families of tasks without structural modification

Families of computational tasks indexed by parameters are incorporated into the model without altering the underlying computational primitives. This allows entire classes of computations to be represented and analyzed within a single structural framework while preserving uniformity of form.

9.7. Partial modification of functions

The model supports localized modification of functions during execution without disrupting the routine structure. Such modifications affect only specific argument pairs and require no changes to the execution order or control mechanisms. This property enables the representation of adaptive and self-modifying computational processes, in the precise sense of localized function updates.

9.8. Minimality of computational primitives

The canonical routine form relies exclusively on binary functions and assignment operations. This minimal set of primitives simplifies formal analysis, supports rigorous proofs, and aligns the model with elementary operational principles commonly used in computational architectures.

9.9. Explicit dependency structure

Each elementary step is fully determined by a fixed set of indices specifying the source values, the target value, and the applied function. This makes all dependencies explicit and renders the computational graph directly analyzable, facilitating structural reasoning and correctness verification.

9.10. Functional–imperative duality

The model establishes a strict and reversible correspondence between compositional (functional) and routine (imperative) representations of computation. This duality allows the advantages of both paradigms to be combined within a single formal framework and provides a bridge between high-level compositional descriptions and low-level operational execution.

Conclusion.

In this paper, a canonical routine form for compositional computation has been introduced through a stepwise structural factorization. The proposed formulation makes execution order, data dependencies, and compositional structure explicit within a unified formal framework. The resulting representation supports compact descriptions, structural analysis, and controlled extensions without increasing model complexity. These properties make the canonical routine form a suitable foundation for further investigations into formal models of computation and their structural properties.

Appendix A. On the Computational Expressiveness of the Canonical Routine Form

This appendix outlines the computational expressiveness of the canonical routine form introduced in the main body of the paper. The purpose of this discussion is not to provide a full proof of computational universality, but to clarify how standard computational mechanisms can be represented within the proposed formalism.

A.1. Memory representation

The canonical routine form operates on a unified global set of values

$$\{xi\}_{i\in I} \subseteq X,$$

which is updated through a sequence of elementary assignment operations. This global state serves as a persistent memory whose contents may be read and modified at each step of the routine. No restrictions are imposed on the reuse of indices, allowing values to be overwritten, propagated, or preserved across arbitrary numbers of steps.

A.2. Elementary operations

Each elementary step of the canonical routine form is represented as an assignment operation acting on the global state of values. In the full routine form, an elementary step has the form

$$x_{T(i)} := g_{K(i)}(x_{S(i)}, x_{L(i)}),$$

where $S(i), L(i)$ and $T(i)$ are indices of elements of the global state, and $K(i)$ specifies the selected binary function from the admissible family.
In the compact formulation with a fixed functional basis, the elementary step takes the simplified form

$$x_{T(i)} := g(x_{S(i)}, x_{L(i)}).$$

Such operations correspond to basic state-update steps in standard imperative computational models. The indices $S(i)$ and $L(i)$ determine the source values used in the computation, while $T(i)$ determines the target location in the global state that is modified by the step. The execution order of these elementary operations is fully determined by the routine index i.

A.3. Conditional control via argument values
As shown in Section 7, the execution order of the routine can be made dependent on the values stored in the global state. By interpreting designated state values as step indices or control parameters, conditional transitions and iterative behavior can be encoded without introducing explicit control-flow instructions.
This mechanism allows the representation of conditional branching and looping constructs as a consequence of state evolution, rather than as separate syntactic entities.

A.4. Iteration and unbounded computation
Since the routine operates over a mutable global state and permits repeated execution of elementary steps under state-dependent control, iterative and potentially unbounded computations can be represented. The absence of structural constraints on the length of the routine or the reuse of state indices allows computations of arbitrary depth to be encoded.

A.5. Relation to standard computational models
The combination of:

- a persistent, mutable memory;
- elementary state-update operations;
- and state-dependent control of execution order

is sufficient to encode the operational behavior of standard imperative computational models, such as register machines or RAM-like abstractions. Consequently, the canonical routine form supports the representation of algorithmic processes with conditional branching, iteration, and state mutation.

A.6. Scope of the expressiveness claim
It is emphasized that this appendix provides a structural correspondence rather than a formal equivalence proof. Establishing full computational universality would require an explicit encoding of a known universal model and a proof of correctness, which lies beyond the scope of the present work.
The intent here is to clarify that the canonical routine form possesses the structural capabilities required to express general algorithmic computation within the framework developed in the paper.

A.7. Concluding remark
The canonical routine form thus provides a compact and structurally explicit representation of computation that aligns with the expressive capabilities of established imperative models, while retaining the advantages of compositional factorization and canonical structure developed in the main text.

Appendix B. Explicit Encoding of a Standard Imperative Model (Sketch)

This appendix provides an explicit sketch of how a standard imperative computational model can be encoded within the canonical routine form introduced in the paper. The purpose is to demonstrate constructively that the proposed formalism is capable of representing conventional algorithmic computation, without aiming at a full universality proof.

B.1. Choice of the reference model

As a reference model, we consider a simple register machine consisting of:

- a finite set of registers storing values from X;
- a finite set of instructions performing elementary updates on registers;
- conditional control based on register values.

This choice is motivated by the close correspondence between register-based computation and the assignment-based structure of the canonical routine form.

B.2. Encoding of registers

Each register of the reference machine is mapped to a distinct element of the global state

$$\{x_i\}_{i\in I} \subseteq \mathrm{X}.$$

used by the canonical routine form.

Register contents are thus represented directly as values of the corresponding state variables. No additional encoding is required, and register updates correspond to assignments to the associated state elements.

B.3. Encoding of instructions

Elementary instructions of the reference register machine are encoded as contiguous segments of the routine consisting of one or more elementary steps of the canonical routine form

$$x_{T(i)} := g_{K(i)}(x_{S(i)}, x_{L(i)}).$$

In the compact formulation with a fixed functional basis, the corresponding elementary step takes the form

$$x_{T(i)} := g(x_{S(i)}, x_{L(i)}).$$

Each instruction of the reference model is represented by a fixed subsequence of routine steps whose structure is determined solely by the indices $S(i), L(i)$, and $T(i)$, together with the routine ordering (and, in the full form, by the selection index $K(i)$).

Basic register operations such as increment, decrement, and copy are represented by choosing appropriate argument indices and constant values stored in designated state locations. Since the functional basis is fixed, all instruction behavior is realized through argument selection and state updates.

B.4. Sequential execution

The program counter of the reference machine is represented implicitly by the routine index i. Each instruction corresponds to a fixed segment of the routine, and the default execution order follows the linear order of the routine steps.

Thus, sequential execution of the register machine is realized directly by the inherent sequentiality of the canonical routine form.

B.5. Conditional branching

Conditional branching is encoded using the state-dependent control mechanism described in Section 7. Specifically, the value of a designated state variable is used to determine the next routine index. By updating this control variable as part of an instruction's execution, conditional jumps are realized without introducing explicit control-flow instructions. Branching behavior arises solely from the evolution of the global state.

B.6. Iteration and loops

Loops in the reference model are encoded by conditional updates that redirect execution to earlier routine indices. Since the routine permits repeated execution of steps under state-dependent control, iterative behavior of arbitrary length can be represented.
No additional looping constructs are required beyond those already present in the canonical routine form.

B.7. Instruction completeness

The encoding described above covers:

- register storage and update;
- sequential instruction execution;
- conditional branching;
- iterative control flow.

These elements constitute the essential operational features of standard imperative models such as register machines. Consequently, the canonical routine form is sufficient to represent their execution behavior.

B.8. Scope and limitations

This appendix provides an explicit constructive correspondence rather than a formal equivalence proof. A full proof of computational universality would require a detailed encoding of a specific universal register machine and a correctness argument for the simulation, which lies beyond the scope of the present work.
The intent of this appendix is to demonstrate concretely that the canonical routine form admits a direct and natural encoding of standard imperative computation.

B.9. Concluding remark

The explicit encoding outlined above complements the structural analysis of computational expressiveness presented in Appendix A. Together, these results show that the canonical routine form is not only structurally expressive but also capable of directly representing conventional algorithmic computation within a compact and formally transparent framework.

Bibliography

- Turing, A. M. (1936)
  On Computable Numbers, with an Application to the Entscheidungsproblem
- Minsky, M. (1967)
  Computation: Finite and Infinite Machines
- Sipser, M.
  Introduction to the Theory of Computation