The Φ–RosettaVM System: A Universal Language Interpreter

Patrick Bernard *December 2025*

Abstract. We present Φ, a meta-language for defining programming languages, paired with RosettaVM, a universal runtime. The key insight is that Φ can describe itself, enabling a bootstrapping tower where Φ(Φ)(L) produces an interpreter for any language L with a Φ specification.

Core Components

Φ	:	Meta-language for defining languages
RosettaVM	:	Universal stack-based runtime
Port	:	Compiler: Φ specs → RosettaVM bytecode

The Φ Operator

$$\Phi : \text{Spec} \times \text{Program} \to \text{Result}$$

Φ(L, p) interprets program p per language spec L. Curried: Φ(L) yields an interpreter for L.

Derived Features

From a specification (sorts, constructors, xforms, rules), Φ derives:

Parser	Grammar from constructors; precedence, layout
Pretty-Printer	Inverse of parsing; configurable formatting
Type Checker	Bidirectional inference from signatures
Interpreter	Direct execution via xform rules
Compiler	Code generation via Port transformation
REPL	Interactive evaluation with introspection

Mathematical Foundations

Initial Algebras	ASTs as μF: fixed points of functors
Yoneda Lemma	$\text{Nat}(F, G) \simeq G(A)$: generic traversals
Cofree Comonads	νF: zippers, lenses, attribute grammars
Recursion Schemes	Cata, ana, hylo, para
Free Monads	$\mu(F + -)$: effect representation
Galois Connections	Xforms as adjunctions: Parse ⊣ Print

Self-Description & Tower

Φ.Φ : Spec — Φ specifies itself, enabling:

Level 0:	RosettaVM (Rust)
Level 1:	Φ in RosettaVM
Level 2:	Φ(Φ.Φ) — Φ interpreting Φ
Level 3:	Φ(Φ.Φ)(L.Φ) — interpreter for L

Implemented Languages

To validate universality, we specify languages across the type theory hierarchy:

λProlog	8KB	Higher-order logic programming with unification
STLC+Nat	8KB	Simply typed λ-calculus with recursion schemes
Poly	9KB	System F: polymorphic lambda calculus
HKT	9KB	Higher-kinded types, Functor/Monad
CoC	17KB	Calculus of Constructions (Coq/Lean core)
Cubical	9KB	Cubical type theory with path types
RVM	11KB	RosettaVM instruction set (self-hosting)
Φ	8KB	Φ itself — meta-circular specification

Each spec produces: parser, type checker, interpreter, and compiler. The CoC and Cubical specs demonstrate that Φ handles dependent types and homotopy type theory.

Execution Paths

Interpret: Φ(L.Φ, prog) → Result

Compile: Port(L.Φ) → L.rvm; RosettaVM.run(L.rvm, prog) → Result

Futamura Projections

proj_1:	Φ(L, p) = result	interpret
proj_2:	Φ(Φ, L) = interp_L	specialize
proj_3:	Φ(Φ, Φ) = compiler	self-apply

Fundamental Theorem

$$\forall L, p : \Phi(\Phi)(L)(p) \equiv \Phi(L)(p) \equiv \text{eval}_L(p)$$

Bootstrap Equation

$$\text{RosettaVM}(\text{Port}(\Phi.\Phi)) \simeq \Phi$$

The compiled Φ spec *is* the Φ operator.

Related Work

Meta-circular interpreters trace to McCarthy's LISP [2], formalized by Reynolds [3]. Our tower adds explicit compilation stages.

Partial evaluation [1, 4] provides the theoretical basis; Port realizes proj_2.

Language workbenches (Spoofax, Rascal, Racket's `#lang`) share our goals but target native compilation.

λProlog [6] demonstrates higher-order logic programming for language specification; Φ draws from this tradition.

Content-addressed code (Unison [7], IPFS) informs RosettaVM's hash-based store.

Stack-based VMs from Forth [8] through WebAssembly inform the instruction set.

Future Directions

- **Full Unification**: Complete Prolog with backtracking
- **Effect Handlers**: Algebraic effects for IO, state, concurrency
- **Content-Addressed Distribution**: P2P code sharing by hash
- **Incremental Compilation**: True partial evaluation (proj_2)
- **Type-Directed Synthesis**: Program synthesis from Φ types
- **Cross-Language Interop**: Zero-cost FFI across the tower

References

[1] Y. Futamura. "Partial Evaluation of Computation Process." *Sys. Comp. Controls* 2(5), 1971.

[2] J. McCarthy. "Recursive Functions of Symbolic Expressions." *CACM* 3(4), 1960.

[3] J.C. Reynolds. "Definitional Interpreters for Higher-Order PLs." *HOSC* 11(4), 1998.

[4] N.D. Jones et al. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[5] H. Abelson, G.J. Sussman. *SICP.* MIT Press, 1996.

[6] D. Miller, G. Nadathur. *Programming with Higher-Order Logic.* Cambridge, 2012.

[7] P. Chiusano, R. Bjarnason. *FP in Scala.* Manning, 2014.

[8] C. Moore. "Forth." *A&A Suppl.* 15, 1974.