Stone: A Monolithically-Typed Programming Language

Gabe Grand
MIT CSAIL
gg@mit.edu
github.com/gabegrand/stone

Abstract

The programming languages community has long been divided by a seemingly intractable debate: static versus dynamic typing, strong versus weak, nominal versus structural. We propose a radical resolution: STONE is a *monolithically-typed* programming language in which the *only* primitive type is **set**. We demonstrate that booleans, natural numbers, ordered pairs, functions, lists, graphs, and finite maps all emerge from pure set theory. By reducing standard language constructs to sets of sets, STONE renders all prior typing debates vacuous. While a naïve implementation of monolithic typing incurs an $O(2^n)$ memory footprint, we present a simple optimization that reduces this to $O(1)$, which we believe to be the largest asymptotic improvement in the history of programming language design. We introduce an efficient, Rust-based implementation and showcase its expressiveness through recursive Fibonacci, primality testing, and the Ackermann function. STONE offers three interchangeable syntax themes—Default, Mathematical, and Zen—the last of which attains a state of notational enlightenment at the modest cost of complete illegibility.

One Type to rule them all, One Type to find them,
One Type to bring them all, and in the emptiness bind them.

—J.R.R. Tolkien

Nothing is certain, but everything is `set` *in stone.*

—Zen kōan

1. Introduction

The history of programming language design is, in large part, a history of arguments about types. Should types be checked at compile time or at run time? Should they be inferred or declared? Should a language have five numeric types or fifty? Should `null` inhabit every type, or should it be banished entirely?

These are important questions. They have occupied some of the finest minds in computer science for over half a century [3, 5]. The resulting landscape is a bewildering taxonomy: *static* vs. *dynamic*, *strong* vs. *weak*, *nominal* vs. *structural*, *manifest* vs. *inferred*, *gradual* vs. *abrupt*. Each axis spawns its own faction, its own conferences, its own flame wars.

We observe that all of these debates share a common assumption: that a language *needs* more than one type. We challenge this assumption.

We present STONE, a programming language with a single primitive type: **set**. Every value in STONE is a finite set. Booleans are sets. Numbers are sets. Pairs are sets. Functions are sets (or, more precisely, are defined *in terms of* sets via comprehensions and closures). The standard library consists of five definitions, all expressible as one-liners. The type system is trivially sound: every well-formed expression produces a set, and every set is well-typed, because there is only one type.

STONE draws its semantic foundations from three results in classical set theory: the Von Neumann construction of ordinals [6], Kuratowski's encoding of ordered pairs [4], and the Zermelo–Fraenkel axioms with Choice [7]. We take these constructions, which are customarily relegated to the foundations of mathematics, and promote them to the *operational semantics* of a real, executable programming language.

Before proceeding, we offer a preview. Figure 1 shows the Fibonacci function written in STONE's Zen notation, in which keywords are replaced by symbols chosen for their spiritual opacity. Below it is the result of computing the fifth Fibonacci number, fully expanded as a Von Neumann ordinal in Zen theme. The result is 5.

```
fib := ▶n. ⫛(n ☯ 0)(0)(
         ⫛(n ☯ 1)(1)(fib(n - 1) + fib(n - 2)))

> fib(5)
{○, ●,
{○, ●},
{○, ●, {○, ●}},
{○, ●, {○, ●}, {○, ●, {○, ●}}}}
```

Figure 1. The fifth Fibonacci number, computed in STONE's Zen notation. In the code, ☯ denotes equality, ▶ denotes lambda, and ⫛ denotes the conditional. In the output, ○ and ● represent zero and one; each layer of nesting is a successor. The result, encoded as a Von Neumann ordinal, is 5.

The remainder of this paper is organized as follows. Section 2 describes the language design, building up from the empty set to a Turing-complete calculus. Section 3 demonstrates applications including Fibonacci, primality testing, data structures, and the Ackermann function. Section 4 presents a performance evaluation, including a novel optimization that achieves an exponential reduction in memory consumption and a rigorous comparison with Python. Section 5 introduces STONE's three display themes and offers meditations on the nature of nothingness. Section 6 concludes.

DOI: http://dx.doi.org/10.1145/(to come)

2. Language Design

STONE programs manipulate a single domain: the universe of hereditarily finite sets. Every value is either the empty set or a finite collection of such values. From this spartan substrate, all familiar programming constructs emerge.

2.1 Booleans

We adopt the standard set-theoretic encoding:

```
false ≔ {}        -- the empty set
true  ≔ {{}}      -- {empty set}
```

Boolean operations are set operations: union is disjunction, intersection is conjunction, and complement is negation.

```
> false \/ true       -- union = OR
1
> true /\ false       -- intersection = AND
0
> !false              -- complement = NOT
1
```

This is not an encoding *of* booleans—it *is* booleans. The set $\{\}$ is falsy because it is empty; $\{\{\}\}$ is truthy because it is not. No further interpretation is required.

2.2 Natural Numbers

We use the Von Neumann ordinals [6], in which each natural number is the set of all smaller natural numbers:

```
> 0                   -- {}
{}
> 1                   -- {{}}
{0}
> 2                   -- {{}, {{}}}
{0, 1}
> 3                   -- {{}, {{}}, {{}, {{}}}}
{0, 1, {0, 1}}
```

Arithmetic operators ($+$, $-$, $\times$, $\div$) are defined on these representations. Subtraction is *monus* (saturating): $3 - 5 = 0$, because negative sets are even more absurd than what we already have.

A delightful consequence of the Von Neumann encoding is that set membership coincides with the less-than relation: $m \in n$ if and only if $m < n$. We exploit this shamelessly:

```
> 2 in 5              -- 2 < 5
true
> 5 in 3              -- 5 < 3
false
```

2.3 Ordered Pairs

Kuratowski's encoding [4] represents the ordered pair (a, b) as the set $\{\{a\}, \{a, b\}\}$:

```
> (1, 2)
(1, 2)
> fst((3, 7))
3
> snd((3, 7))
7
```

The projection functions, despite their apparent simplicity, are among the most satisfying definitions in the standard library:

```
fst ≔ fn p. Union Intersection p
snd ≔ fn p. Union{x in Union p :
        !(x in Intersection p)
        ∪ (Union p = Intersection p)}
```

The reader is invited to verify these on paper. We recommend setting aside an afternoon.

2.4 Conditionals and Recursion

STONE has no `if`/`then`/`else` construct. Instead, conditional selection emerges from set comprehensions:

```
cond ≔ fn b. fn x. fn y.
  Union{x : _ in b} ∪ Union{y : _ in !b}
```

When b is true ($= \{\{\}\}$, non-empty), the first comprehension yields x and the second is empty. When b is false ($= \{\}$, empty), the converse holds. Combined with lambda expressions (`fn x. expr`) and lazy evaluation of `cond` arguments, this gives us a Turing-complete language.

2.5 Set Comprehensions

Comprehensions are STONE's workhorse. Map, filter, and Cartesian product are all expressed with a single syntactic form:

```
> {x + 1 : x in 5}            -- map
{1, 2, 3, 4, 5}
> {x in 10 : x / 2 * 2 == x} -- filter
{0, 2, 4, 6, 8}
> {(a,b) : a in {1,2}, b in {3,4}}
{(1, 3), (1, 4), (2, 3), (2, 4)}
```

2.6 The Standard Library

The complete STONE standard library consists of five definitions, reproduced here in their entirety:

```
cond ≔ fn b. fn x. fn y.
  Union{x : _ in b} ∪ Union{y : _ in !b}
succ ≔ fn n. n ∪ {n}
pred ≔ fn n.
  Union{m in n : m ∪ {m} = n}
fst  ≔ fn p. Union Intersection p
snd  ≔ fn p.
  Union{x in Union p :
    !(x in Intersection p)
    ∪ (Union p = Intersection p)}
```

This is the entire runtime. There are no imports, no modules, no package manager, no supply chain attacks. The dependency graph is the empty set.

3. Applications

We now demonstrate that STONE is capable of expressing non-trivial computations, each of which would typically require multiple types in a conventional language.

3.1 Fibonacci

```
fib ≔ fn n. cond(n = 0)(0)(
  cond(n = 1)(1)(
    fib(n - 1) + fib(n - 2)))

> fib(10)
55
```

We note that `55` is, of course, a set with 55 elements, each of which is itself a set. The fully expanded representation of `fib(10)` contains several thousand nested braces. We omit it for the sake of the typesetter.

3.2 Primality Testing

The Von Neumann membership trick enables an elegant primality test. Since $1 \in d$ iff $d > 1$, we can filter candidate divisors without explicit comparison operators:

```
mod ≔ fn a. fn b. a - (a / b) * b

is_prime ≔ fn n. cond(1 in n)(
  {d in n : cond(1 in d)(
```

```
    mod(n)(d) = 0)(false)} = empty
  )(false)

primes := fn N. {p in N : is_prime(p)}

> primes(20)
{2, 3, 5, 7, 11, 13, 17, 19}
```

Every prime number shown above is a set. The set of primes is a set of sets. The power set of the primes below 20 has $2^8 = 256$ elements, each a set of sets of sets. We advise against computing it.

3.3 Data Structures

Since ordered pairs are sets, we can encode standard data structures directly.

Lists. A list is either the empty set (`nil`) or a Kuratowski pair of a head and tail:

```
nil  := empty
cons := fn h. fn t. (h, t)
head := fn l. fst(l)
tail := fn l. snd(l)
len  := fn l.
  cond(l = nil)(0)(1 + len(tail(l)))

> cons(1)(cons(2)(cons(3)(nil)))
(1, (2, (3, 0)))
```

Finite maps. A map is a set of key-value pairs with lookup by filtering:

```
m      := {(1, 10), (2, 20), (3, 30)}
lookup := fn m. fn k.
  Union{snd(e) : e in m, fst(e) = k}

> lookup(m)(2)
20
```

Graphs. A directed graph is a set of edge-pairs:

```
edges     := {(1,2), (2,3), (3,4), (1,3)}
neighbors := fn v.
  {snd(e) : e in edges, fst(e) = v}

> neighbors(1)
{2, 3}
```

3.4 Relation Composition

Composition of binary relations—a matter for databases or proof assistants—is a one-liner:

```
R := {(1,2), (2,3)}
S := {(2,4), (3,5)}
compose := {(fst(r), snd(s)) :
  r in R, s in S, snd(r) = fst(s)}

> compose
{(1, 4), (2, 5)}
```

3.5 The Ackermann Function

To demonstrate that STONE is capable of computing extremely large sets, we implement the Ackermann function [1]:

```
ack := fn m. fn n.
  cond(m = 0)(n + 1)(
    cond(n = 0)(ack(pred(m))(1))(
      ack(pred(m))(ack(m)(pred(n)))))

> ack(3)(2)
29
```

Benchmark	STONE	Python	Slowdown
`fib(10)`	9 ms	0.008 ms	1,100×
`fact(6)`	3 ms	0.0003 ms	10,000×
`primes(20)`	3 ms	0.005 ms	600×
`ack(3,4)`	475 ms	0.5 ms	950×
Type errors	0	∞	—

Table 1. Benchmark results comparing STONE and Python 3. Both implementations use naïve recursion. The final row reports a metric on which STONE achieves optimal performance.

The value 29 is a Von Neumann ordinal containing 29 elements. The computation of $\mathrm{ack}(4)(2)$ is left as an exercise for the reader's patience and electricity bill.

4. Performance

A natural concern with monolithic typing is performance. Under the Von Neumann encoding, the natural number n is represented as a set containing 2^n nodes when fully expanded: $0 = \varnothing$ requires one node, $1 = \{\varnothing\}$ requires two, $2 = \{\varnothing, \{\varnothing\}\}$ requires four, and in general n requires 2^n. The result of $\mathrm{fib}(10) = 55$ would, if naïvely materialized as a Von Neumann set tree, occupy approximately 2^{55} nodes—roughly 36 petabytes of memory at one byte per node. This is impractical on most currently available hardware.

4.1 A Novel Optimization

We observe that the exponential blowup can be entirely eliminated by storing Von Neumann naturals as native 64-bit integers. Under this representation, the natural number n is stored as the machine word n, reducing the memory footprint from $O(2^n)$ to $O(1)$. We believe this to be the largest asymptotic improvement in the history of programming language implementation.

Concretely, STONE's Rust runtime represents each natural number as a `Nat(u64)` variant, bypassing the set-tree representation entirely. Arithmetic operations dispatch to native hardware instructions. The theoretical semantics remain set-theoretic; only the representation has changed. This is, of course, standard practice in every other programming language ever designed, but we present it here as a contribution.

4.2 Empirical Evaluation

To assess the practical consequences of monolithic typing, we benchmark STONE against Python 3, both using naïve recursive implementations without memoization. All benchmarks were executed on a single core of an Apple M2 processor. Results are shown in Table 1.

Several observations are in order.

First, STONE is between three and four orders of magnitude slower than Python across all computational benchmarks. We consider this a remarkable result. Given that every integer in STONE is, semantically, a tower of nested sets—and that every function application involves closure construction, environment capture, and set-theoretic conditional dispatch—the fact that the overhead is *merely* polynomial rather than exponential validates the optimization described in Section 4.

Second, the Ackermann benchmark is the most computationally honest of the four, as its 475 ms execution time renders startup overhead negligible. The 950× slowdown thus reflects the true interpretive overhead of set-theoretic computation, uncontaminated by fixed costs.

Third, and most importantly, STONE achieves *zero* type errors across all benchmarks. This is a provable invariant of the language:

```
is_prime := ▶n. ⫛(1 ∈ n)(
  {d ∈ n : ⫛(1 ∈ d)
    (mod(n)(d) ◕ 0)(○)}
  ◕ ○)(○)
```

Figure 2. Primality testing in Zen notation. The reader who can parse this without consulting the symbol table has achieved *satori*.

since there is exactly one type, no program can produce a type error, regardless of its inputs. Python, by contrast, offers no such guarantee, and the potential for type errors at runtime is, in principle, unbounded. We report this as ∞ in Table 1, acknowledging that the precise value is implementation-dependent.

We leave closing the remaining performance gap to future work, though we note that correctness is, as the saying goes, priceless.

5. The Zen of Stone

STONE offers three display themes, reflecting the philosophical journey from pragmatism through formalism to enlightenment.

Default. The default theme, for the practical-minded:

```
fib ≔ fn n. cond(n = 0)(0)(
  cond(n = 1)(1)(fib(n-1) + fib(n-2)))
```

Mathematical. Unicode mathematical notation, for the mathematically inclined:

```
fib := λn. cond(n = 0)(0)(
  cond(n = 1)(1)(fib(n-1) + fib(n-2)))
```

Zen. For those who have transcended the need for comprehension:

```
fib := ▶n. ⫛(n ◕ 0)(0)(
  ⫛(n ◕ 1)(1)(fib(n-1) + fib(n-2)))
```

In Zen theme, λ becomes ▶, equality becomes ◕, the conditional becomes ⫛, and the empty set becomes ○. True is ●. Successor is ↑; predecessor, ↓. Programs in Zen notation are visually striking and semantically opaque, achieving the ideal balance for publication in prestigious venues.

Figure 2 shows the primality test in Zen theme, offered without further explanation.

5.1 Kōans

We close this section with meditations for the aspiring STONE programmer.

1. *Nothing is certain, but everything is set in stone.*
2. *The empty set is false. The set containing the empty set is true. Thus, truth is the embrace of nothingness.*
3. *The student asked: "What is the type of a set?" The master replied: "Set." The student asked: "And the type of that set?" The master struck the student with the empty set. The student was enlightened. Also, unhurt.*
4. *To know a number is to contain all that is less than it. Five is not merely five; it is zero, and one, and two, and three, and four, holding each within itself. This is the Von Neumann way.*
5. *A language with one type has no type errors. A language with no type errors needs no type checker. A language with no type checker compiles instantly. Thus,* STONE *achieves zero-cost abstractions by having no abstractions.*

6. Conclusion

We have presented STONE, a programming language that resolves the decades-long debate over type systems by the simple expedient of having exactly one type. STONE demonstrates that booleans, natural numbers, ordered pairs, conditionals, recursion, lists, maps, graphs, and relations can all be expressed as pure set-theoretic constructions—not as encodings *into* a host language, but as the language itself.

Our standard library consists of five one-line definitions. Our type system is trivially sound. Our Zen theme is beautiful and unreadable. We believe these are features.

Future work. We plan to extend STONE with infinite sets (requiring only a resolution of the Continuum Hypothesis), networking support (encoding TCP packets as Von Neumann ordinals), and a package manager (which will distribute packages as power sets of dependency sets, inheriting the 2^n blowup that modern ecosystems have already achieved in practice). Additionally, we plan to write an operating system in STONE. The kernel will be the empty set.

Availability. STONE is implemented in Rust, compiles to native code, and is available under an open-source license. The source code, all examples shown in this paper, and the Zen symbol chart can be found at https://github.com/gabegrand/stone.

References

[1] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[4] C. Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1):161–171, 1921.

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

[6] J. Von Neumann. Zur Einführung der transfiniten Zahlen. *Acta Scientiarum Mathematicarum (Szeged)*, 1:199–208, 1923.

[7] E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908.