Why: Explicit Regions Without Annotations, Between C and Go

Baptiste Boussemart
Noematic
contact@noematic.eu

Abstract

Every language imposes a trade-off: C offers performance but leaves memory management to the programmer, Go offers simplicity, especially for concurrent programs, at the cost of a garbage collector, and Rust offers safety at the cost of lifetimes written in signatures. **Why** is a native language prototype that aims for the performance of C and the simplicity of Go for multithreaded programs, with no garbage collector and no `free`. The programmer names a lexical block, `arena doc { ... }`, reclaimed at the closing brace; a guard is a struct that contains its own arena and moves from call to call. Across a call, the compiler infers which parameter is stored into which other: signatures carry no region variable. This paper presents the model, the analysis, and a first account of the measurements: 76 regression programs pass, and the median Why/C ratio over 46 integer kernels is about 1.01. A longer paper details each of the announced studies.

1 Introduction

Motivation. No language excels everywhere. C gives a control and a performance few languages match, but every `malloc` awaits its `free`, and forgetting or doubling one is a fault the compiler does not see. Go makes concurrent programming simple, with lightweight tasks, channels, and a sober syntax, but relies on a garbage collector whose cost and pauses are outside the programmer's control. Rust achieves safety without a collector, but asks the programmer to write lifetimes in types and signatures. Why starts from a simple frustration: getting the performance of C and the simplicity of Go for multithreaded programs. That goal rules out a garbage collector, and it also rules out a return to C's manual memory management.

Regions as an answer. Many programs have collective deaths: a document discarded at reload, a frame at the end of a tick, a request at the end of its handling. Regions name that death directly. Cyclone showed it, with pointer types that carry the name of their region [Grossman et al. 2002]. Why asks a more precise question:

> How far can one build an explicit-region language, without region annotations on pointer types, by inferring storage relations across functions?

```
func load() error {
  arena doc {
    buf := read(doc) or return
    ls := parse(doc, buf)
    show(ls)
  }
  return nil
}
```

`buf` and `ls` are allocated in `doc`. At the closing brace the compiler inserts one call that reclaims the chunks of `doc`; the author writes no `free`. If `load` tried to return a slice into `buf`, the checker would reject that return and name the value that would escape. The type `*T` carries no region: the compiler deduces, for each function, which parameter is stored into which other.

Contributions.

1. A programming model that combines three lexical regions, movable guards, and tasks joined by their block (§2).
2. An interprocedural analysis of storage effects, with no region annotation in signatures, whose fixed point is bounded (§3).
3. A first account of the costs and limits of the existing compiler (§4), and the programme of coming studies (§6).

2 The model in brief

Three regions. Every Why value lives in one of three regions: the *call stack*, reclaimed when the function returns; the *named* `arena` *block*, reclaimed at the closing brace by `why_arena_free`; and the *task arena* `arena thread`, for what must outlive the function without a named block, reclaimed when the task returns. The rule the examples follow is to allocate where the consumer lives: the caller passes the arena, because the caller knows which block should own the result.

Guards: structs that contain an arena. Some

results must outlive the block that built them. A struct marked `garde` is an ordinary value, on the stack or passed by value, that contains its own arena: `t.make` and `t.append` allocate into it. `return` and `=` move the value, and the arena follows; the source is reset to zero. When the name that holds it dies, the compiler reclaims that inner arena like an `arena` block; if the value was moved, that reclamation does nothing.

```
struct Buf garde {
  xs []int
}

func build() Buf {
  arena doc {
    t := Buf
    t.xs = t.make([]int, 1)
    t.xs[0] = 7
    return t
  }
}

func main() int {
  b := build()
  return b.xs[0]
}
```

This program is accepted and exits 7. The same shape with an ordinary struct, whose slice was allocated by `doc.make`, is rejected at the `return`: `the slice does not escape`. The ordinary struct points into storage of `doc`, which dies at the brace; the guard contains its arena and takes it along. A guard can be neither read after a move (`moved guard`) nor copied (`the guard does not copy`), because two copies would reclaim the same arena twice.

The escape rule (E). *A reference does not outlive the region that owns its storage.* If a slice, a map, a pointer, an interface, or a guard is born in region `R` and the program stores it, returns it, or passes it to a name that lives longer than `R`, the checker rejects that shape. The author chooses allocation regions in the text; the compiler infers only the storage *effects* across calls.

Aliasing and freezing. A local rule complements (E): in a block, a write through a pointer is rejected when another address of the same variable is live (`mutable pointer`). `freeze` marks a name and rejects the writes identified on it; making it transitive is work in progress.

Concurrency. `thread f(x)` submits a task to a pool, and the block that launched it joins it before reclaiming its arena. Channels `chan[T](n)` are bounded queues. Passing a block arena to a task is refused (`the arena does not escape`): a task cannot use storage its launcher is about to reclaim. The programmer writes `thread` and `chan` as in Go, and the death stays attached to the brace.

$$\frac{\texttt{destination.xs = source} \;\Rightarrow\; S(\texttt{store}) = \{(\texttt{destination}, \texttt{source})\}}{\texttt{store(destination, source)} \;\Rightarrow\; S(\texttt{relay}) = \{(\texttt{destination}, \texttt{source})\}}$$

$$\frac{(p, q) \in S(f) \qquad \text{life}(a_p) \not\supseteq \text{life}(\text{reg}(a_q))}{f(a_1, \ldots, a_n) \text{ rejected}}$$

Figure 1: Summaries of the example and the call rule. $\text{reg}(a_q)$ is the region that owns the storage of the stored argument; the call is refused if the destination a_p can outlive it.

3 Storage effects across calls

The summary (S(f)) of a function is a set of pairs `(Ptr, From)`: `Ptr` is the receiver or a parameter that receives a store, `From` the parameter or receiver that is stored. At the call, each pair is confronted with the actual arguments (Figure 1).

Fixed point. Direct stores seed the summaries. A function that forwards a call inherits the callee's pairs whose two ends project onto its own parameters. Iteration recomposes the calls until a pass adds no pair; since the set of pairs is finite, it terminates. A ceiling of 64 passes additionally fixes the compilation budget: beyond it, compilation fails and no binary is written.

Example. `relay` only forwards, yet it inherits the store performed by `store`, and the call is refused in `main` because `xs` is born in an arena shorter than that of `h`. The same relay within a single arena is accepted (exit 7).

```
struct Hold { xs []int }

func store(destination *Hold, source []int) {
  destination.xs = source
}

func relay(destination *Hold, source []int) {
  store(destination, source)
}

func main() int {
  arena outer {
    h := Hold
    arena inner {
      xs := inner.make([]int, 1)
      xs[0] = 7
      relay(h, xs)      // the slice does not escape
    }
    return h.xs[0]
```

```
  }
}
```

What the author does not write. In Rust, the same program requires a named lifetime on the struct and in both signatures; the elision rules [Rust Reference] do not suffice, because they give each input reference a distinct lifetime.

```
struct Hold<'a> { xs: &'a [i64] }
fn store<'a>(destination: &mut Hold<'a>,
             source: &'a [i64]) {
    destination.xs = source;
}
fn relay<'a>(destination: &mut Hold<'a>,
             source: &'a [i64]) {
    store(destination, source);
}
```

Cyclone likewise requires a region in every pointer type, and explicit outlives constraints in the prototype when two regions differ [Cyclone Manual]. Why rejects the same `main` with signatures identical to those of a program without regions.

The price: modularity. Inferring from the body has a counterpart. The summary is part of a function's contract without being written: changing the body of `store` can change which calls are accepted elsewhere, and checking a call requires the callee's body or its summary. Cyclone precisely fixes the meaning of omitted annotations in prototypes to allow separate compilation [Cyclone Manual]. Why makes the opposite choice and partly compensates for it: a summary is a small set of pairs, which a module interface file could export and a tool could display. An `extern` declaration has no body; its summary is empty and the C is trusted. An interface call requires the summaries of its implementations to agree (otherwise `the methods do not agree`).

4 Translation and first results

Why emits C11 and invokes `cc -std=c11 -fwrapv -fno-strict-aliasing -O2`; C serves as both ABI and intermediate representation. The runtime fits in one 934-line header: bump allocation, 4096-byte chunks recycled through a thread-local pool, a pool of `min(cpus, 8)` workers, and bounded channels. On leaving a block, the emitted code closes channels, joins tasks, runs `defer`s, reclaims guards, then calls `why_arena_free`.

On integer code, the emitted C is on par with hand-written C: that is the first half of the goal. The region-allocation cycle still costs 1.3 to 1.9 times `malloc/free`, which points to the allocator as the first optimisation target. The second half of the goal, simplicity, and the benefit of regions on frames, graphs, or irregular lifetimes remain to be measured (§6). There is no proof of (E) yet.

Question	Observation
Does the checker refuse the forbidden stores?	76 regression programs pass, expected rejections included.
Is the emitted C on par with C?	Median Why/C ≈ 1.01 over 46 integer kernels; notable outlier on map update ($\approx 6\times$).
What does a region-allocation cycle cost?	1.33× against `malloc/free` for a pooled 4 KB batch; 1.4–1.9× for 64 KB freed; peak memory close to C.
Do real programs compile?	A file server and a TLS client/server compile and return the expected code.

Table 1: First account of the measurements. Protocols are detailed in the longer paper.

5 Positioning

Cyclone is the closest published system: it already has the region block, but its pointer types carry the region and its signatures are region-polymorphic [Grossman et al. 2002, Jim et al. 2002]. Tofte and Talpin, and later the ML Kit, infer the *placement* of values in regions [Tofte and Talpin 1994, Tofte et al. 2004]; Why leaves placement to the author and infers only storage effects. Rust states lifetimes in signatures, and RustBelt proves the safety of a realistic fragment [Jung et al. 2018]. Verona organises objects into isolated regions and controls aliasing with reference capabilities, with a soundness proof [Arvidsson et al. 2023]; Hylo avoids lifetime annotations through mutable value semantics [Racordon et al. 2022]. Why takes another path: storable references, without annotation, checked by inferred summaries. Zig's explicit allocators [Zig] and library arenas such as bumpalo [Fitzgerald] leave the choice to the call site; in Why the block is mandatory and the escape check is part of the language. Go, whose syntax Why borrows, leaves death to a garbage collector, which remains preferable when nobody wants to name the death of a graph.

6 The coming studies

The longer paper, to appear, develops the following points.

- **The full model and its proof.** Pointers without `&`, strings and errors, reclamation order, and a proof of (E) for a closed fragment, with a transitive `freeze`.

- **The interprocedural analysis.** The effect of traversal order, interface calls, exporting summaries for separate compilation, and an inventory of safe programs refused.
- **Runtime and concurrency.** The bounded pool and its deadlock risk, channels, `arena shared`, and sharing of mutable data between threads, currently outside the checked perimeter.
- **Evaluation.** Annotations avoided compared with Rust and Cyclone on the same programs, a frame loop, a graph in a region, irregular lifetimes, and allocator optimisation.
- **Simplicity.** Measuring the "on par with Go" goal on concurrent programs written in both languages.

7 Conclusion

Why aims for the performance of C and the simplicity of Go for multithreaded programs, with no garbage collector and no `free`. It keeps allocation regions explicit and infers the storage effects that cross a call, without region annotations on `*T`. The checker refuses a store, a return, or a relay when the inferred pair crosses into a shorter region. The prototype compiles real programs and is on par with C on integer code; the announced studies must now show that annotation-free regions keep the promise of simplicity.

Availability. Compiler, test suite, and benchmarks: https://git.noematic.eu/baptiste/why and https://git.noematic.eu/baptiste/why-comparaison-benchmark.

References

Arvidsson, E., Castegren, E., Clebsch, S., Drossopoulou, S., Noble, J., Parkinson, M. J., and Wrigstad, T. Reference capabilities for flexible memory management. *PACMPL* 7, OOPSLA2 (2023), article 270. DOI `10.1145/3622846`.

Cyclone Manual. Cyclone User's Manual, chapter "Memory Management Via Regions". https://www.cs.cornell.edu/Projects/cyclone/online-manual/main-screen008.html.

Fitzgerald, N. bumpalo, a bump allocator for Rust. https://github.com/fitzgen/bumpalo, accessed November 2026.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in Cyclone. In *PLDI 2002*, pp. 282–293. DOI `10.1145/512529.512563`.

Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. Cyclone: a safe dialect of C. In *USENIX ATC 2002*, pp. 275–288.

Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. RustBelt: securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), article 66. DOI `10.1145/3158154`.

Racordon, D., Shabalin, D., Zheng, D., Abrahams, D., and Saeta, B. Implementation strategies for mutable value semantics. *Journal of Object Technology* 21, 2 (2022). DOI `10.5381/jot.2022.21.2.a2`.

Rust Reference. Lifetime elision. https://doc.rust-lang.org/reference/lifetime-elision.html, accessed November 2026.

Tofte, M. and Talpin, J.-P. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL 1994*, pp. 188–201. DOI `10.1145/174675.177855`.

Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation* 17, 3 (2004), pp. 245–265. DOI `10.1023/B:LISP.0000029446.78563.a4`.

Zig. Zig Language Reference, section "Memory". https://ziglang.org/documentation/master/#Memory, accessed November 2026.