Why : des régions explicites sans annotation, entre C et Go

Baptiste Boussemart
Noematic
contact@noematic.eu

Résumé

Chaque langage impose un compromis : C offre la performance mais laisse au programmeur la gestion de la mémoire, Go offre la simplicité, notamment pour les programmes concurrents, mais au prix d'un ramasse-miettes, et Rust offre la sûreté au prix de durées de vie écrites dans les signatures. **Why** est un prototype de langage natif qui vise la performance du C et la simplicité de Go pour des programmes multithreadés, sans ramasse-miettes et sans `free`. Le programmeur nomme un bloc lexical, `arena doc { ... }`, libéré à l'accolade fermante ; une garde est une structure qui contient sa propre arène et se déplace d'appel en appel. À travers un appel, le compilateur infère quel paramètre est stocké dans quel autre : les signatures ne portent aucune variable de région. Cet article présente le modèle, l'analyse et un premier état des mesures : 76 programmes de non-régression passent, et la médiane des rapports Why/C sur 46 noyaux de calcul entier est d'environ 1,01. Une version longue détaille chacune des études annoncées.

1 Introduction

Motivation. Aucun langage n'excelle partout. C donne un contrôle et une performance que peu de langages égalent, mais chaque `malloc` attend son `free`, et l'oubli ou le doublon de l'un d'eux est une faute que le compilateur ne voit pas. Go rend la programmation concurrente simple, avec des tâches légères, des canaux et une syntaxe sobre, mais repose sur un ramasse-miettes dont le coût et les pauses échappent au programmeur. Rust obtient la sûreté sans ramasse-miettes, mais demande d'écrire des durées de vie dans les types et les signatures. Why part d'une frustration simple : obtenir la performance du C et la simplicité de Go pour des programmes multithreadés. Cet objectif exclut le ramasse-miettes, et il exclut aussi le retour à la gestion manuelle du C.

Les régions comme réponse. Beaucoup de programmes ont des fins de vie collectives : un document abandonné au rechargement, une trame à la fin d'un pas de temps, une requête à la fin de son traitement. Les régions nomment directement cette fin de vie. Cyclone l'a montré, avec des types pointeurs qui portent le nom de leur région [Grossman et al. 2002]. Why pose une question plus précise :

> Jusqu'où peut-on construire un langage à régions explicites, sans annotations de région sur les types pointeurs, en inférant les relations de stockage entre fonctions ?

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

`buf` et `ls` sont alloués dans `doc`. À l'accolade fermante, le compilateur insère un seul appel qui libère les blocs mémoire de `doc` ; l'auteur n'écrit aucun `free`. Si `load` tentait de renvoyer une tranche sur `buf`, le vérificateur rejetterait ce retour en nommant la valeur qui s'échapperait. Le type `*T` ne porte aucune région : le compilateur déduit, pour chaque fonction, quel paramètre est stocké dans quel autre.

Contributions.

1. Un modèle de programmation combinant trois régions lexicales, des gardes déplaçables et des tâches attendues par leur bloc (§2).
2. Une analyse interprocédurale des effets de stockage, sans annotation de région dans les signatures, dont le point fixe est borné (§3).
3. Un premier état des coûts et des limites du compilateur existant (§4), et le programme des études à venir (§6).

2 Le modèle en bref

Trois régions. Toute valeur Why vit dans l'une de trois régions : la *pile d'appels*, libérée au retour de la fonction ; le *bloc* ***`arena`*** *nommé*, libéré à l'accolade fermante par `why_arena_free` ; l'*arène de tâche* `arena thread`, pour ce qui doit survivre à la fonction sans bloc

nommé, libérée au retour de la tâche. La règle suivie par les exemples est d'allouer là où vit le consommateur : l'appelant passe l'arène, car c'est lui qui sait quel bloc doit posséder le résultat.

Les gardes : des structures qui contiennent une arène. Certains résultats doivent survivre au bloc qui les a construits. Une structure marquée `garde` est une valeur ordinaire, sur la pile ou passée par valeur, qui contient sa propre arène : `t.make` et `t.append` y allouent. `return` et `=` déplacent la valeur, et l'arène suit ; la source est remise à zéro. Lorsque le nom qui la détient cesse de vivre, le compilateur libère cette arène interne comme un bloc `arena` ; si la valeur a été déplacée, cette libération ne fait rien.

```
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]
}
```

Ce programme est accepté et se termine avec le code 7. La même forme avec une structure ordinaire, dont la tranche est allouée par `doc.make`, est rejetée au `return` : `la tranche ne sort pas`. La structure ordinaire pointe vers une mémoire de `doc`, qui meurt à l'accolade ; la garde contient son arène et l'emporte. Une garde ne peut être ni relue après déplacement (`garde déplacée`) ni copiée (`la garde ne se copie pas`), car deux copies libéreraient deux fois la même arène.

La règle d'échappement (E). *Une référence ne survit pas à la région qui possède sa mémoire.* Si une tranche, une table associative, un pointeur, une interface ou une garde naît dans la région `R` et que le programme la stocke, la renvoie ou la transmet à un nom qui vit plus longtemps que `R`, le vérificateur rejette cette forme. L'auteur choisit les régions d'allocation dans le texte ; le compilateur infère seulement les *effets* de stockage à travers les appels.

Aliasing et gel. Une règle locale complète (E) : dans un bloc, une écriture à travers un pointeur est rejetée lorsqu'une autre adresse de la même variable est vivante (`pointeur mutable`). `freeze` marque un nom et rejette les écritures identifiées sur lui ; le rendre transitif est un travail en cours.

```
destination.xs = source
  ⇒ S(store) = {(destination, source)}
store(destination, source)
  ⇒ S(relay) = {(destination, source)}
```

$$\frac{(p,q) \in S(f) \qquad \text{vie}(a_p) \not\supseteq \text{vie}(\text{rég}(a_q))}{f(a_1, \ldots, a_n) \text{ rejeté}}$$

Figure 1: Résumés de l'exemple et règle d'appel. $\text{rég}(a_q)$ est la région qui possède la mémoire de l'argument stocké ; l'appel est refusé si la destination a_p peut lui survivre.

Concurrence. `thread f(x)` soumet une tâche à un pool, et le bloc qui l'a lancée l'attend avant de libérer son arène. Les canaux `chan[T](n)` sont des files bornées. Passer une arène de bloc à une tâche est refusé (`l'arène ne sort pas`) : une tâche ne peut pas utiliser une mémoire que son lanceur va libérer. Le programmeur écrit `thread` et `chan` comme en Go, et la fin de vie reste attachée à l'accolade.

3 Effets de stockage à travers les appels

Le résumé (S(f)) d'une fonction est un ensemble de paires `(Ptr, From)` : `Ptr` est le receveur ou un paramètre qui reçoit un stockage, `From` le paramètre ou le receveur stocké. À l'appel, chaque paire est confrontée aux arguments effectifs (Figure 1).

Point fixe. Les stockages directs initialisent les résumés. Une fonction qui relaie un appel hérite des paires de l'appelé dont les deux extrémités se projettent sur ses propres paramètres. L'itération recompose les appels jusqu'à une passe qui n'ajoute aucune paire ; l'ensemble des paires étant fini, elle termine. Un plafond de 64 passes fixe en plus le budget de compilation : au-delà, la compilation échoue et aucun binaire n'est écrit.

Exemple. `relay` ne fait que transmettre ; il hérite pourtant du stockage de `store`, et l'appel est refusé dans `main` parce que `xs` naît dans une arène plus courte que `h`. Le même relais dans une seule arène est accepté (code 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)      // la tranche ne sort pas
    }
    return h.xs[0]
  }
}
```

Ce que l'auteur n'écrit pas. En Rust, le même programme exige une durée de vie nommée sur la structure et dans les deux signatures ; les règles d'élision [Rust Reference] ne suffisent pas, car elles attribuent une durée distincte à chaque référence en entrée.

```
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 demande de même une région dans chaque type pointeur, et des contraintes « survit à » explicites dans le prototype lorsque deux régions diffèrent [Cyclone Manual]. Why rejette le même `main` avec des signatures identiques à celles d'un programme sans régions.

Le prix : la modularité. Inférer depuis le corps a une contrepartie. Le résumé fait partie du contrat d'une fonction sans être écrit : modifier le corps de `store` peut changer les appels acceptés ailleurs, et vérifier un appel suppose de connaître le corps de l'appelé ou son résumé. Cyclone fixe justement le sens des annotations omises dans les prototypes pour permettre la compilation séparée [Cyclone Manual]. Why fait le choix inverse et le compense en partie : un résumé est un petit ensemble de paires, qu'un fichier d'interface de module pourrait exporter et qu'un outil pourrait afficher. Une déclaration `extern` n'a pas de corps ; son résumé est vide et le code C fait foi. Un appel d'interface exige que les résumés de ses implémentations concordent (sinon `les méthodes ne s'accordent pas`).

Question	Observation
Le vérificateur refuse-t-il les stockages interdits ?	76 programmes de non-régression passent, rejets attendus compris.
Le C émis est-il au niveau du C ?	Médiane Why/C $\approx 1{,}01$ sur 46 noyaux de calcul entier ; écart notable sur la mise à jour de table associative ($\approx 6\times$).
Que coûte un cycle d'allocation par région ?	1,33× face à `malloc`/`free` pour un lot de 4 Ko mis en pool ; 1,4 à 1,9× pour 64 Ko libérés ; pic mémoire proche du C.
Des programmes réels compilent-ils ?	Un serveur de fichiers et un client/serveur TLS compilent et renvoient le code attendu.

Table 1: Premier état des mesures. Protocoles détaillés dans la version longue.

4 Traduction et premiers résultats

Why émet du C11 et invoque `cc -std=c11 -fwrapv -fno-strict-aliasing -O2` ; le C sert d'ABI et de représentation intermédiaire. L'environnement d'exécution tient dans un en-tête de 934 lignes : allocation par avancement de curseur, blocs de 4096 octets recyclés dans un pool local au thread, pool de `min(cpus, 8)` threads de travail et canaux bornés. À la sortie d'un bloc, le code émis ferme les canaux, attend les tâches, exécute les `defer`, libère les gardes, puis appelle `why_arena_free`.

Sur du calcul entier, le C émis est au niveau du C écrit à la main : c'est la première moitié de l'objectif. Le cycle d'allocation par région coûte encore entre 1,3 et 1,9 fois `malloc`/`free`, ce qui désigne l'allocateur comme premier chantier d'optimisation. La seconde moitié de l'objectif, la simplicité, et le bénéfice des régions sur des trames, des graphes ou des durées de vie irrégulières restent à mesurer (§6). Il n'existe pas encore de preuve de (E).

5 Positionnement

Cyclone est le système publié le plus proche : il possède déjà le bloc de région, mais ses types pointeurs portent la région et ses signatures sont polymorphes en régions [Grossman et al. 2002, Jim et al. 2002]. Tofte et Talpin, puis le ML Kit, infèrent le *placement* des valeurs dans les régions [Tofte et Talpin 1994, Tofte et al. 2004] ; Why laisse le placement à l'auteur et n'infère que les effets de stockage. Rust énonce les durées de vie dans les signatures, et RustBelt prouve la sûreté d'un fragment réaliste [Jung et al. 2018]. Verona organise les objets

en régions isolées et contrôle l'aliasing par des capacités de référence, avec une preuve de correction [Arvidsson et al. 2023] ; Hylo évite les annotations de durée de vie par la sémantique de valeur mutable [Racordon et al. 2022]. Why choisit une autre voie : des références stockables, sans annotation, vérifiées par des résumés inférés. Les allocateurs explicites de Zig [Zig] ou les arènes en bibliothèque comme bumpalo [Fitzgerald] laissent le choix au site d'appel ; dans Why, le bloc est obligatoire et le contrôle d'échappement fait partie du langage. Go, dont Why reprend la syntaxe, confie la fin de vie au ramasse-miettes, qui reste préférable lorsque personne ne veut nommer celle d'un graphe.

6 Les études à venir

La version longue, à paraître, développe les points suivants.

- **Le modèle complet et sa preuve.** Pointeurs sans `&`, chaînes et erreurs, ordre de libération, et une preuve de (E) sur un fragment fermé, avec un `freeze` transitif.
- **L'analyse interprocédurale.** Effet de l'ordre de parcours, appels d'interface, export des résumés pour la compilation séparée, et inventaire des programmes sûrs refusés.
- **L'exécution et la concurrence.** Pool borné et risque d'interblocage, canaux, `arena shared`, et partage de données mutables entre threads, aujourd'hui hors du périmètre vérifié.
- **L'évaluation.** Annotations évitées par rapport à Rust et Cyclone sur les mêmes programmes, boucle de trames, graphe dans une région, durées de vie irrégulières, et optimisation de l'allocateur.
- **La simplicité.** Mesurer l'objectif « au niveau de Go » sur des programmes concurrents écrits dans les deux langages.

7 Conclusion

Why vise la performance du C et la simplicité de Go pour des programmes multithreadés, sans ramasse-miettes et sans `free`. Il garde des régions d'allocation explicites et infère les effets de stockage qui traversent un appel, sans annotation de région sur `*T`. Le vérificateur refuse un stockage, un retour ou un relais lorsque la paire inférée franchit une région plus courte. Le prototype compile des programmes réels et atteint le niveau du C sur du calcul entier ; les études annoncées doivent maintenant montrer que les régions sans annotation tiennent la promesse de simplicité.

Disponibilité. Compilateur, suite de tests et bancs d'essai : https://git.noematic.eu/baptiste/why et https://git.noematic.eu/baptiste/why-comparaison-benchmark.

Références

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

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

Fitzgerald, N. bumpalo, un allocateur linéaire pour Rust. https://github.com/fitzgen/bumpalo, consulté en novembre 2026.

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

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

Jung, R., Jourdan, J.-H., Krebbers, R., et 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., et 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, consulté en novembre 2026.

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

Tofte, M., Birkedal, L., Elsman, M., et 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, consulté en novembre 2026.