

Engineer

Rust

Langage compilé vérifiant les règles de sûreté mémoire



Ce que c’est

Rust est un langage de programmation destiné au travail traditionnellement réalisé en C et C++, où le programme gère lui-même sa mémoire et s’intègre souvent à des logiciels écrits dans d’autres langages. Son compilateur produit du code machine natif avant l’exécution. La version 1.0, annoncée le 15 mai 2015, associait un contrôle de bas niveau des performances à des garanties de sûreté obtenues sans ramasse-miettes ni environnement d’exécution. C’est ce qui permet à une bibliothèque Rust de remplacer une bibliothèque C.

La mémoire est gérée par la propriété, un ensemble de règles vérifiées par le compilateur. Chaque valeur possède un seul propriétaire à la fois. Lorsque celui-ci quitte sa portée, la valeur est détruite et sa mémoire libérée, à un moment fixé lors de la compilation plutôt qu’au prochain passage d’un ramasse-miettes. Le code qui a besoin d’une valeur sans en prendre possession en emprunte une référence. Les règles d’emprunt autorisent à tout moment soit une référence modifiable, soit un nombre quelconque de références non modifiables, et imposent la validité de chaque référence. Un programme qui enfreint une règle ne compile pas. Ces mêmes règles excluent les courses aux données (data races) : deux threads qui écrivent sur les mêmes données sans synchronisation provoquent donc une erreur de compilation plutôt qu’une panne intermittente.

Rust n’a pas d’exceptions. Une fonction pouvant échouer de manière attendue renvoie un `Result`, contenant soit une valeur, soit une erreur. Le compilateur émet par défaut un avertissement si l’appelant laisse le résultat inutilisé. Un défaut irrécupérable, comme une lecture au-delà de la fin d’un tableau, passe plutôt par la macro `panic!`.

L’analyse statique est prudente : le compilateur rejette donc certains programmes qui auraient fonctionné correctement. Certains travaux, comme le pilotage du matériel ou les appels à un autre langage, ne peuvent pas du tout être vérifiés. Pour ces cas, `unsafe` donne accès à cinq capacités supplémentaires, dont le déréférencement d’un pointeur brut et l’appel d’une fonction non sûre, ce qui inclut par défaut toute fonction déclarée depuis C. Il ne désactive pas le vérificateur d’emprunts : il marque les lignes où le programmeur, plutôt que le compilateur, garantit la sûreté mémoire. La sûreté de ces opérations dépend donc du respect de leurs contrats par le programmeur, y compris dans le code sûr qui les entoure. Le livre Rust recommande de garder ces blocs courts et de les entourer d’une interface sûre, comme le font certaines parties de la bibliothèque standard.

Un nouveau compilateur paraît toutes les six semaines. Les changements susceptibles de casser le code existant attendent une édition, publiée environ tous les trois ans. Il en existe quatre : 2015, 2018, 2021 et 2024. La dernière est devenue stable le 20 février 2025 avec Rust 1.85.0. Chaque paquet indique son édition dans `Cargo.toml`, chaque compilateur compile le code de toutes les éditions qui l’ont précédé et les paquets d’éditions différentes peuvent être liés entre eux. Un projet passe donc à une nouvelle édition quand il le décide. Cargo, gestionnaire de dépendances et outil de construction, est fourni avec le langage. rustfmt harmonise le style du code, et les bibliothèques tierces arrivent sous forme de paquets appelés crates.

Utilisez le concept pour prendre une décision

Utilisez « Qu’est-ce que Rust ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c’est important

En C et C++, un pointeur utilisé après la libération de sa mémoire ou deux threads écrivant sur les mêmes données compilent sans protestation et échouent ensuite de manière intermittente, souvent loin de la cause. Rust rejette ces programmes en signalant des erreurs de compilation. Puisque les vérifications ont lieu pendant la construction, le programme final n’en paie aucun coût.

Sans ramasse-miettes ni environnement d'exécution à embarquer, le code Rust peut servir de socle à des logiciels écrits dans d'autres langages. Tout langage capable d'appeler C peut appeler un cœur Rust par une interface C : une implémentation bien testée peut donc servir des clients sur plusieurs plateformes. Le livre Rust cite parmi les usages en production les outils en ligne de commande, les services web, les appareils embarqués et d'importantes parties du navigateur Firefox.

Le coût se paie en amont. Il faut structurer le code pour que le compilateur puisse en constater la sûreté, ce qui oblige souvent à décider qui possède une valeur plutôt qu'à transmettre librement des références. Le livre lui-même précise qu'il faut du temps pour s'habituer à la propriété. Lorsque le compilateur refuse un programme qu'il ne peut pas vérifier, la réponse consiste généralement à en modifier la structure plutôt qu'à contourner le contrôle.

La stabilité est l'autre partie du contrat. Depuis la version 1.0, les changements incompatibles ont été largement exclus : une modification susceptible de casser l'ancien code n'arrive donc que par une édition qu'un paquet choisit d'adopter.

À quoi cela ressemble en pratique

Dans le portfolio, Rust constitue le cœur d'un client de messagerie natif pour macOS : le cœur bien testé du protocole est écrit en Rust et expose une interface C. Le client, 11 141 lignes de Swift réparties en 53 fichiers, le lie sous forme d'archive statique compilée depuis une révision amont fixée. Ce choix compte, car l'interface C constitue le contrat : un cœur qui évolue modifie ce contrat sans changer une ligne de Swift. Chaque élément renvoyé par l'interface C s'accompagne d'une règle de propriété. Une erreur produit une fuite ou un plantage sans avertissement intermédiaire du compilateur. Une fine couche d'encapsulation possède donc chaque pointeur et chaque chaîne traversant la frontière, et aucune couche supérieure ne voit de pointeur brut.

Lier ce cœur à une application signifie livrer toutes ses dépendances. C'est le sujet d'un audit de licences de 644 crates Rust. Chaque crate déclare sa licence sous forme d'expression. À chaque construction, l'audit vérifie toutes les expressions du graphe complet de dépendances selon une liste d'autorisation : OR représente un choix entre licences, AND l'obligation de respecter les deux. Toute expression non reconnue fait échouer la construction. Un contrôle n'ayant jamais échoué ressemble à un contrôle incapable d'échouer : celui-ci a donc été volontairement mis en échec deux fois, d'abord en remplaçant la licence d'une

crate par un terme refusé, puis en changeant la révision fixée du cœur sans régénérer l'audit. Il ne lit que les métadonnées déclarées et ne constitue pas un avis juridique.

Ce site n'a pas de code source Rust propre. Il rencontre ce langage indirectement, par Ruff, lui-même écrit en Rust, qui vérifie chaque script Python touché par un commit.

Où les problèmes surviennent

La plupart des difficultés avec Rust se situent aux limites des vérifications du compilateur : arithmétique dans une construction optimisée, frontière avec un autre langage et code apporté par une dépendance.

L'arithmétique reboucle dans une version de production. Un compteur qui provoquait une panique par dépassement pendant les tests devient silencieusement un petit nombre en production. Dans les versions de débogage, Rust vérifie les dépassements arithmétiques des entiers et provoque une panique. Une version construite avec `--release` omet le contrôle et reprend à l'autre extrémité de la plage des valeurs : 256 dans un `u8` devient donc 0. Le livre considère comme une erreur le fait de compter sur ce comportement. Lorsqu'une valeur risque de dépasser sa capacité, choisissez explicitement le comportement : `checked_*` renvoie `None`, `saturating_*` s'arrête à la limite et `wrapping_*` reboucle délibérément.

```
fn add_stock(on_hand: u8, delivered: u8) -> Option<u8> {
    // None when the sum does not fit, in debug and release alike
    on_hand.checked_add(delivered)
}
```

Les défauts mémoire reviennent à la frontière avec C. Un programme dont la partie Rust se construit sans le moindre avertissement présente une fuite de mémoire ou plante lors d'un appel étranger. L'appel de fonctions déclarées dans un bloc `extern` est considéré comme non sûr, car l'autre langage n'impose aucune règle de Rust. Une convention invisible au compilateur détermine donc qui libère chaque pointeur. Confiez tous les pointeurs et toutes les chaînes traversant cette frontière à une même couche d'encapsulation, documentez chaque règle de propriété à côté de l'appel et n'exposez au-dessus qu'une interface sûre.

Une panique atteint C et le processus s'interrompt. Une fonction Rust appelée depuis C provoque une panique. Au lieu de renvoyer une erreur, tout le programme s'arrête. Depuis Rust 1.81, paru le 5 septembre 2024, une panique tentant de sortir d'une fonction déclarée avec l'interface simple `"C"` interrompt le processus. Transformez les échecs en valeurs de retour avant qu'ils atteignent la frontière et ne déclarez une fonction avec `"C-unwind"` que si le code en face s'attend à un dépilage.

Une mise à jour ordinaire change la licence de l'ensemble du logiciel. Un changement de version à trois niveaux de profondeur apporte une crate dont la licence est inacceptable pour le produit, sans que la construction le signale. Chaque crate déclare sa propre licence, et les obligations s'additionnent sur tout le graphe. Vérifiez chaque expression selon une liste d'autorisation à chaque construction, faites échouer toute expression non reconnue et provoquez volontairement un échec une fois pour prouver que le contrôle réagit.

Pour aller plus loin

- Le langage de programmation Rust — le livre officiel, de l'installation à la propriété, à la gestion des erreurs et au code non sûr.
- Qu'est-ce que la propriété ? — les trois règles de propriété et la libération de la mémoire sans ramasse-miettes.
- Rust non sûr — les cinq capacités non sûres, les pointeurs bruts et les appels C par `extern`.
- Le format du manifeste — les champs `edition`, `rust-version` et `license` de `Cargo.toml`.
- Annonce de Rust 1.0 — les objectifs initiaux et la promesse de stabilité commencée avec la version 1.0.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que Rust ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/rust/