Engineer

Rust

Kompilierte Sprache mit Prüfung der Speichernutzung

Was es ist

Rust ist eine Programmiersprache für Aufgaben, die traditionell in C und C++ erledigt werden: Programme verwalten ihren Speicher selbst und werden oft mit Software in anderen Sprachen verknüpft. Der Compiler erzeugt vorab nativen Maschinencode. Version 1.0 wurde am 15. Mai 2015 als Sprache angekündigt, die hardwarenahe Kontrolle über die Leistung mit Sicherheitsgarantien verbindet, ohne Garbage Collector und ohne Laufzeitumgebung. Das erlaubt einer Rust-Bibliothek, eine C-Bibliothek zu ersetzen.

Speicher wird durch Besitzregeln verwaltet, die der Compiler prüft. Jeder Wert hat zu einem Zeitpunkt genau einen Besitzer. Verlässt dieser seinen Gültigkeitsbereich, wird der Wert verworfen und sein Speicher freigegeben. Der Zeitpunkt steht beim Kompilieren fest, statt davon abzuhängen, wann ein Garbage Collector das nächste Mal nachsieht. Code, der einen Wert braucht, ohne ihn zu übernehmen, leiht sich eine Referenz aus. Die Ausleihregeln erlauben zu jedem Zeitpunkt entweder eine veränderbare Referenz oder beliebig viele unveränderbare und verlangen, dass jede Referenz gültig ist. Ein Programm, das eine Regel verletzt, kompiliert nicht. Dieselben Regeln schließen Datenrennen aus. Wenn zwei Threads ohne Synchronisation in dieselben Daten schreiben, führt das deshalb zu einem Compilerfehler statt zu einem sporadischen Defekt.

Rust hat keine Ausnahmen. Eine Funktion, die auf erwartbare Weise scheitern kann, gibt ein `Result` zurück, das entweder einen Wert oder einen Fehler enthält. Der Compiler warnt standardmäßig, wenn der Aufrufer das Ergebnis ungenutzt lässt. Ein Defekt, von dem sich das Programm nicht erholen kann, etwa ein Zugriff hinter das Ende eines Arrays, führt stattdessen über das Makro `panic!` .

Statische Analyse ist konservativ. Deshalb weist der Compiler manche Programme zurück, die durchaus funktioniert hätten. Andere Aufgaben, etwa Hardware anzusteuern oder eine andere Sprache aufzurufen, lassen sich überhaupt nicht prüfen. Dafür gibt es `unsafe`, das fünf zusätzliche Fähigkeiten freischaltet, darunter das Dereferenzieren eines Rohzeigers und den Aufruf einer unsicheren Funktion. Dazu gehört standardmäßig jede als extern deklarierte C-Funktion. Es schaltet den Borrow-Checker nicht aus. Es markiert die Zeilen, in denen der Programmierer statt des Compilers für Speichersicherheit einsteht, sodass die Sicherheit dieser Operationen davon abhängt, dass der Programmierer ihre Verträge auch im umgebenden sicheren Code einhält. Das Rust-Buch rät, diese Blöcke klein zu halten und mit einer sicheren Schnittstelle zu umgeben. So sind Teile der Standardbibliothek aufgebaut.

Alle sechs Wochen erscheint ein neuer Compiler. Änderungen, die vorhandenen Code brechen würden, warten auf eine Edition, die ungefähr alle drei Jahre erscheint. Es gibt vier: 2015, 2018, 2021 und 2024. Die neueste wurde am 20. Februar 2025 mit Rust 1.85.0 stabil. Jedes Paket nennt seine Edition in `Cargo.toml`, jeder Compiler baut alle zuvor erschienenen Editionen, und Pakete verschiedener Editionen lassen sich zusammenlinken. Ein Projekt wechselt deshalb dann, wenn es möchte. Cargo, die Abhängigkeitsverwaltung und das Buildwerkzeug, gehört zur Sprache. rustfmt hält den Codestil einheitlich, und Bibliotheken Dritter kommen als Pakete namens Crates.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist Rust?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Der Teil, den Nutzer nie sehen: wo die Daten liegen, welche Regeln gelten und wie andere Systeme darauf zugreifen dürfen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

6 ETL/ELT-Pipelines entworfen und betrieben Kontakt aufnehmen →

Warum es wichtig ist

In C und C++ kompiliert ein Zugriff über einen Zeiger nach der Speicherfreigabe ebenso klaglos wie das Schreiben zweier Threads in dieselben Daten. Der Fehler zeigt sich später, sporadisch und oft weit von seiner Ursache entfernt. Rust weist solche Programme mit Compilerfehlern zurück. Weil die Prüfungen beim Build laufen, kosten sie das fertige Programm nichts.

Ohne Garbage Collector und mitzuführende Laufzeitumgebung kann Rust-Code als Grundlage für Software in anderen Sprachen dienen. Alles, was C aufrufen kann, kann über eine C-Schnittstelle auch einen Rust-Kern aufrufen. Eine gut getestete Implementierung kann deshalb Clients auf mehreren Plattformen bedienen. Das Rust-Buch nennt Kommandozeilenwerkzeuge, Webdienste, eingebettete Geräte und wesentliche Teile des Firefox-Browsers als produktive Einsatzgebiete.

Der Preis fällt im Voraus an. Code muss so gestaltet sein, dass der Compiler seine Sicherheit erkennen kann. Oft bedeutet das, den Besitzer eines Werts festzulegen, statt Referenzen frei weiterzureichen. Das Buch selbst sagt, dass man Zeit braucht, um sich an die Besitzregeln zu gewöhnen. Lehnt der Compiler ein Programm ab, das er nicht verifizieren kann, ist meist eine andere Struktur die Antwort, kein Umweg um die Prüfung.

Stabilität ist die andere Hälfte des Geschäfts. Seit 1.0 sind inkompatible Änderungen weitgehend ausgeschlossen. Eine Änderung, die alten Code brechen könnte, kommt deshalb nur mit einer Edition, für die sich ein Paket ausdrücklich entscheidet.

Wie es in der Praxis aussieht

Im Portfolio bildet Rust den Kern unter einem nativen macOS-Nachrichtenclient. Der gut getestete Protokollkern ist in Rust geschrieben und stellt eine C-Schnittstelle bereit. Der Client mit 11.141 Swift-Zeilen in 53 Dateien linkt ihn als statisches Archiv, kompiliert aus einer festgeschriebenen Revision des Ursprungsprojekts. Das Festschreiben ist wichtig, weil die C-Schnittstelle der Vertrag ist: Ein sich ändernder Kern ändert den Vertrag, ohne eine einzige Swift-Zeile zu ändern. Alles, was die C-Schnittstelle zurückgibt, hat eine Besitzregel. Wird eine falsch behandelt, entstehen ein Speicherleck oder ein Absturz, ohne dass der Compiler dazwischen warnt. Deshalb besitzt eine dünne Hüllschicht jeden Zeiger und jede Zeichenkette, die die Grenze überqueren. Keine Schicht darüber sieht einen Rohzeiger.

Den Kern in eine Anwendung zu linken heißt, alle seine Abhängigkeiten mitzuliefern. Darum geht es bei einer Lizenzprüfung von 644 Rust-Crates. Jede Crate deklariert ihre Lizenz als Ausdruck. Die Prüfung vergleicht bei jedem Build jeden Ausdruck im vollständigen Abhängigkeitsgraphen mit einer Positivliste. OR gilt dabei als Wahl zwischen Lizenzen, AND als Verpflichtung, beide einzuhalten. Alles nicht Erlaubte lässt den Build scheitern. Eine Prüfung, die noch nie

fehlgeschlagen ist, sieht aus wie eine, die es nicht kann. Deshalb wurde sie zweimal absichtlich zum Scheitern gebracht: einmal durch Umschreiben einer Crate-Lizenz auf einen von der Liste abgelehnten Begriff, einmal durch Ändern der festgeschriebenen Kernrevision, ohne das Audit neu zu generieren. Sie liest ausschließlich deklarierte Metadaten und ist keine Rechtsberatung.

Diese Website hat keinen eigenen Rust-Quelltext. Sie begegnet der Sprache mittelbar über Ruff, das in Rust geschrieben ist und jedes von einem Commit berührte Python-Skript prüft.

Wo es schiefgeht

Die meisten Rust-Probleme liegen an den Grenzen der Compilerprüfung: Arithmetik in einem optimierten Build, die Grenze zu einer anderen Sprache und Code, der mit einer Abhängigkeit kommt.

Ganzzahlen springen im Release-Build nach einem Überlauf zurück. Ein Zähler, der im Test bei einem Überlauf eine Panic auslöste, wird in Produktion unbemerkt zu einer kleinen Zahl. In Debug-Builds prüft Rust Ganzzahlarithmetik auf Überläufe und löst eine Panic aus. Ein Build mit `--release` lässt diese Prüfung weg und setzt die Rechnung nach einem Überlauf am anderen Ende des Wertebereichs fort, sodass 256 in einem `u8` zu 0 wird. Das Buch behandelt es als Fehler, sich darauf zu verlassen. Wo ein Wert überlaufen kann, wählen Sie das Verhalten ausdrücklich: `checked_*` gibt `None` zurück, `saturating_*` bleibt an der Grenze stehen, und `wrapping_*` verwendet dieses zyklische Verhalten ausdrücklich.

```
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)
}
```

Speicherfehler kommen an der C-Grenze zurück. Ein Programm, dessen Rust-Teil sich ohne eine einzige Warnung bauen lässt, weist Speicherlecks auf oder stürzt beim Aufruf einer Funktion in einer anderen Sprache ab. Der Aufruf von Funktionen aus einem `extern` -Block gilt als unsicher, weil die andere Sprache keine Rust-Regeln durchsetzt. Wer welchen Zeiger freigibt, ist deshalb eine für den Compiler unsichtbare Konvention. Geben Sie einer Hüllschicht die

Verantwortung für jeden übergebenen Zeiger und jede Zeichenkette, schreiben Sie jede Besitzregel neben den Aufruf und bieten Sie darüber nur eine sichere Schnittstelle an.

Eine Panic erreicht C, und der Prozess bricht ab. Eine von C aufgerufene Rust-Funktion löst eine Panic aus. Statt eines Fehlers endet das gesamte Programm. Seit Rust 1.81, veröffentlicht am 5. September 2024, bricht eine Panic, die eine mit der gewöhnlichen Schnittstelle `"C"` deklarierte Funktion verlassen will, den Prozess ab. Wandeln Sie Fehler vor der Grenze in Rückgabewerte um und deklarieren Sie eine Funktion nur dann mit `"C-unwind"`, wenn der Code auf der anderen Seite ein Abwickeln des Stacks erwartet.

Ein gewöhnliches Update ändert die Lizenz des ganzen Builds. Ein Versionssprung drei Ebenen tiefer zieht eine Crate mit einer für das Produkt nicht akzeptablen Lizenz herein, ohne dass der Build etwas meldet. Jede Crate deklariert ihre eigene Lizenz. Die Verpflichtungen summieren sich über den ganzen Graphen. Prüfen Sie bei jedem Build jeden Crate-Ausdruck gegen eine Positivliste, lassen Sie alles nicht Passende scheitern und bringen Sie die Prüfung einmal absichtlich zum Scheitern, um ihre Wirksamkeit zu belegen.

Weiterführende Lektüre

- Die Programmiersprache Rust — Das offizielle Buch von der Installation über Besitzregeln und Fehlerbehandlung bis zu unsicherem Code.
- Was ist Ownership? — Die drei Besitzregeln und die Speicherfreigabe ohne Garbage Collector.
- Unsicheres Rust — Die fünf unsicheren Fähigkeiten, Rohzeiger und Aufrufe von C über `extern`.
- Das Manifestformat — Die Felder `edition`, `rust-version` und `license` in `Cargo.toml`.
- Rust 1.0 angekündigt — Die ursprünglichen Ziele und das mit 1.0 begonnene Stabilitätsversprechen.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist Rust?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/rust/