Test et CI de projets Rust

Hadrien Grasland 2024-09-24

- Langage sur un **créneau similaire à C++**
  - Compilation AoT, pas de GC, contrôle bas niveau
  - Typage strict/explicite, méta-prog', *zero-cost abstraction*
  - Effort d'apprentissage important

Du langage Rust...

- Langage sur un **créneau similaire à C++**
  - Compilation AoT, pas de GC, contrôle bas niveau
  - Typage strict/explicite, méta-prog', *zero-cost abstraction*
  - Effort d'apprentissage important
- ...mais avec des différences importantes
  - Brise la compatibilité pour **plus d'ergonomie et de sûreté**
  - **Excellent outillage** en standard (*build*, deps, tests, doc...)

…à sa perception*

- Il y a **du travail pour que ça compile**…

* Paraphrase libre de nombreux retours d'expérience d'utilisateurs du langage.

…à sa perception*

- Il y a **du travail pour que ça compile**…
- …mais **ensuite ça marche très bien**
  - Peu d’erreurs à l’exécution
  - Facile à faire évoluer
  - Performances stables et élevées
  - « Prévisible » et « ennuyeux » en prod

* Paraphrase libre de nombreux retours d’expérience d’utilisateurs du langage.

…à sa perception*

- Il y a **du travail pour que ça compile**…
- …mais **ensuite ça marche très bien**
  - Peu d'erreurs à l'exécution
  - Facile à faire évoluer
  - Performances stables et élevées
  - « Prévisible » et « ennuyeux » en prod
- Plus qu'à accélérer la compilation ?

* Paraphrase libre de nombreux retours d'expérience d'utilisateurs du langage.

Si ça compile, ça marche ?

```
fn main() {
    assert_eq!(2 + 2, 5);
}
```

- Compile sans erreur ⟶ *Crash* à l'exécution
- Zut, toujours pas de compilateur omniscient…

Si ça compile, ça marche ?

```
fn main() {
    assert_eq!(2 + 2, 5);
}
```

- Compile sans erreur → *Crash* à l'exécution
- Zut, toujours pas de compilateur omniscient…

- Mais alors, d'où vient la réputation de fiabilité de Rust ?
  - **Conception moins piégeuse** du langage et API std/crates
  - **Culture de la qualité** + **outils** pour l'assurer

Si ça compile, ça marche ?

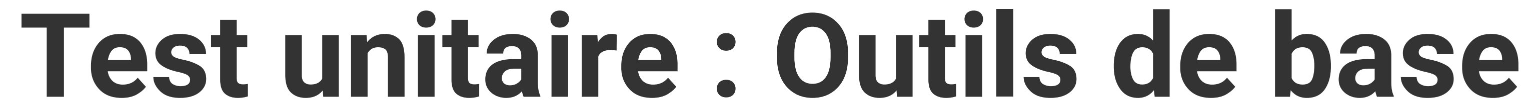
```rust
fn main() {
    assert_eq!(2 + 2, 5);
}
```

- Compile sans erreur → *Crash* à l'exécution
- Zut, toujours pas de compilateur omniscient…
- Mais alors, d'où vient la réputation de fiabilité de Rust ?
  - **Conception moins piégeuse** du langage et API std/crates
  - **Culture de la qualité** + **outils** pour l'assurer

Analyse statique*

- **Formatage** : cargo fmt
- **Nettoyage dépendances** : cargo machete
- **Logique du code** : Lints rustc + cargo clippy
  - Explorer aussi les *lints* « allow-by-default »
- **Liens internes de la doc API** : cargo doc
- **Respect de SemVer** : cargo semver-check

* Ici, je parle des analyseurs de code non modifiés, la preuve formelle arrive plus loin



- Avant : **Assertions** dans l'implémentation (debug_ si besoin)
  - Alternative : Empêcher l'erreur au niveau type/API

Test unitaire : Outils de base

- Avant : **Assertions** dans l'implémentation (debug_ si besoin)
  - Alternative : Empêcher l'erreur au niveau type/API
- Pendant : Privilégier le ***property-based testing*** avec proptest
  - Biaiser le RNG pour équilibrer (ex : valide vs invalide)
  - Garder des tests oracle pour les cas tordus
  - Attention aux erreurs rares (*mocking* si besoin)

Test unitaire : Outils de base

- Avant : **Assertions** dans l'implémentation (debug_ si besoin)
  - Alternative : Empêcher l'erreur au niveau type/API
- Pendant : Privilégier le ***property-based testing*** avec proptest
  - Biaiser le RNG pour équilibrer (ex : valide vs invalide)
  - Garder des tests oracle pour les cas tordus
  - Attention aux erreurs rares (*mocking* si besoin)
- Après : Vérifier la **couverture de code** avec tarpaulin

Test unitaire : Aller plus loin

- **Vérifier si les tests échouent** quand on change…
  - …l'implémentation des méthodes : cargo mutants
  - …l'ordonnancement des *threads* : shuttle
  - …la disponibilité réseau (système distribué) : turmoil

Test unitaire : Aller plus loin

- **Vérifier si les tests échouent** quand on change…
  - …l'implémentation des méthodes : cargo mutants
  - …l'ordonnancement des *threads* : shuttle
  - …la disponibilité réseau (système distribué) : turmoil
- Parfois, on peut être **exhaustif**
  - *Brute force* : Paramètre bool, entier/flottant ≤ 32-bit…
  - Ordonnancement des *threads* : loom
  - Vérification formelle : Kani, Prusti, Verus, MIRAI, Creusot…

Autres types de tests

- **Exemples de doc API :** Inclus dans cargo test !
- **Beaucoup de tests d'intégration ?** Essayez cargo nextest
  - Exécution plus parallèle que cargo test
  - …mais ne supporte pas encore les doctests

Autres types de tests

- **Exemples de doc API :** Inclus dans cargo test !
- **Beaucoup de tests d'intégration ?** Essayez cargo nextest
  - Exécution plus parallèle que cargo test
  - …mais ne supporte pas encore les doctests
- **Exemples de manuels mdbook :** Essayez mdbook test
  - Quand ça ne suffit pas, {{#include …}} + test normal

Analyse dynamique pour *unsafe*

- **Code « isolé »** (ni FFI ni syscalls) : cargo miri
  - Vérifie presque tous les UBs possibles avec *unsafe*
  - …mais lent + Rust seulement + modèle mémoire incomplet

Analyse dynamique pour *unsafe*

- **Code « isolé »** (ni FFI ni syscalls) : cargo miri
  - Vérifie presque tous les UBs possibles avec *unsafe*
  - …mais lent + Rust seulement + modèle mémoire incomplet
- Sinon, **cargo careful avec *sanitizers***
  - careful : Assertions de sûreté supplémentaires std/core
  - Sanitizers : Erreurs communes à C et Rust *unsafe*

Et si tout ça ralentit ma CI ?

- D'abord, **optimiser sa compilation**
  - Outils : cargo build --timings, cargo llvm-lines, measureme…
  - Lire attentivement Tips for Faster Rust Compile Times

Et si tout ça ralentit ma CI ?

- D'abord, **optimiser sa compilation**
  - Outils : cargo build --timings, cargo llvm-lines, measureme…
  - Lire attentivement Tips for Faster Rust Compile Times
- **Cache des dépendances** compilées, voire toolchain Rust
  - setup-rust-toolchain sur GitHub, sinon sccache

Et si tout ça ralentit ma CI ?

- D’abord, **optimiser sa compilation**
  - Outils : cargo build --timings, cargo llvm-lines, measureme…
  - Lire attentivement Tips for Faster Rust Compile Times
- **Cache des dépendances** compilées, voire toolchain Rust
  - setup-rust-toolchain sur GitHub, sinon sccache
- **Paralléliser paralléliser paralléliser**
  - …mais attention aux redondances dans les matrices

Ex petit projet : ~3min + arrêt rapide

Triggered via push last month
HadrienG2 pushed 08a58a1 scheduled-semver-checks

Status
Success

Total duration
2m 58s

ci.yml
on: push

matrix_vars 0s

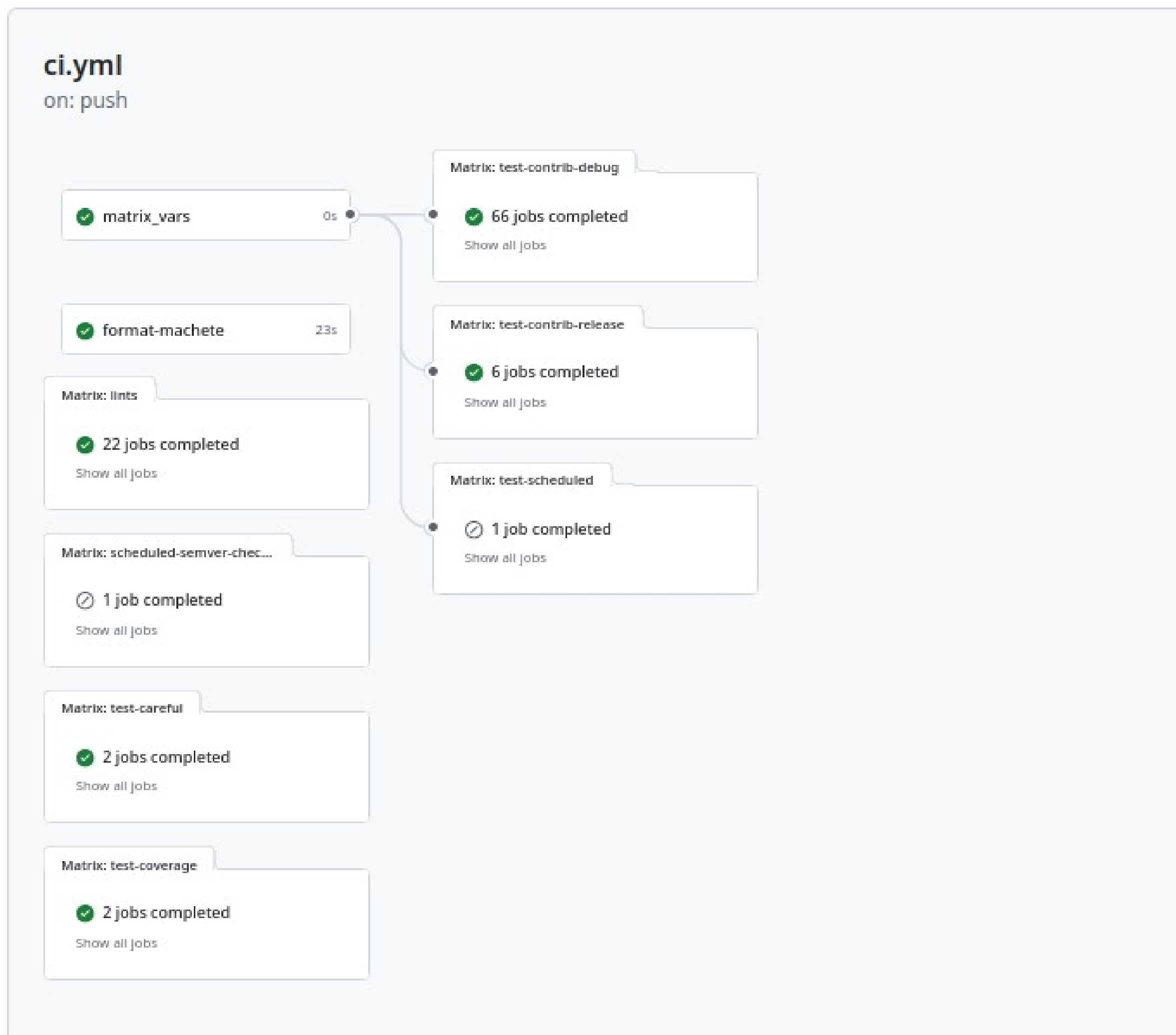
format-lints 2m 48s
scheduled-semver-checks 0s

Matrix: test-contrib
test-contrib (macos-latest... 59s
test-contrib (macos-lat... 1m 20s
test-contrib (ubuntu-lates... 58s
test-contrib (ubuntu-lat... 1m 2s
test-contrib (windows-l... 1m 45s
test-contrib (windows-l... 1m 59s

Matrix: test-scheduled
1 job completed
Show all jobs

- format ~10s
- clippy/rustc +15s
- semver +2m30s
- En parallèle, tests
  - OS : win/mac/linux
  - rustc : stable/min

Ex projet plus gros : <15min possible



- hwlocality = 43 kLOCs
- 11 configurations à tester
  - …sous 3 OS
  - …avec 2 versions rustc
- Et en bonus…
  - Analyseurs statiques
  - Cargo careful + sanitizers
  - Couverture de code
  - Test de *build* release

Et en dehors de la CI push/PR ?

- **CI mensuelle** avec rustc beta + nightly + minimum pour…
  - …contribuer au test des nouveaux compilateurs
  - …corriger les lints avant qu’elles arrivent en stable
  - …détecter les incompatibilités deps vs rustc minimum

Et en dehors de la CI push/PR ?

- **CI mensuelle** avec rustc beta + nightly + minimum pour…
  - …contribuer au test des nouveaux compilateurs
  - …corriger les lints avant qu'elles arrivent en stable
  - …détecter les incompatibilités deps vs rustc minimum
- **Dependabot** ou équivalent pour gérer les dépendances
  - Les maintenir à jour (si Cargo.lock dans le dépôt)
  - Etre informé de problèmes sécu (ex : abandon)

Comment tester la perf ?

- Globalement, **difficile en CI** sans un nœud dédié
  - Si nécessaire, métriques non temporelles (ex : instructions)
  - Des progrès côté visualisation, cf infra rustc, Bencher

Comment tester la perf ?

- Globalement, **difficile en CI** sans un nœud dédié
  - Si nécessaire, métriques non temporelles (ex : instructions)
  - Des progrès côté visualisation, cf infra rustc, Bencher
- En attendant, **pour un benchmarking local** solide
  - criterion = Mesures de *throughput* avec stats rigoureuses
  - Attention, parfois la latence, conso RAM… sont importants
  - Couvrir les cas courants ET les cas tordus intéressants

Conclusion

- Le langage/compilo n’est qu’un contributeur à la qualité
  - Ergonomie qui évite voire empêche certaines erreurs
- Rust doit aussi beaucoup à sa communauté
  - Forte préoccupation qualité des gros projets
  - Beaucoup d’outils puissants prêts à l’emploi
  - Et je n’ai pas parlé des *guidelines*, tutoriels…
- Si vous codez en Rust, faites honneur à sa réputation :)

Merci pour votre attention !

Éviter les erreurs d’utilisation

- **Conventions de nommage :** sleep_duration > x, param, run()
- **Types spécifiques (*newtypes*) :** Duration > i32, f32
- **Etats à la compilation (*typestates*) :** FooBuilder → Foo
- **Enums > bool, entiers**
  - Types algébriques → Paramètres inter-dépendants !
- **Result<T, E>** force l’appelant à gérer l’erreur E
  - Compléter avec une doc des cas d’erreurs