ENS Paris-Saclay – L3 Informatique
Programmation Avancée
Projet de programmation

Jacques-Henri Jourdan & Armaël Guéneau

2025

1 Présentation générale

L'objectif de ce projet est d'implémenter un « borrow checker » pour un sous-ensemble idéalisé du langage de programmation Rust. Le borrow checker est un composant crucial du compilateur Rust : avec le vérificateur de types, ces deux composants garantissent la sûreté du langage. Si un programme est accepté à la fois par le vérificateur de types et le borrow checker, il est garanti qu'il ne présentera aucun comportement indéfini.

En Rust, le vérificateur de types garantit la *sûreté des types* : les valeurs manipulées par le programme ont toujours le type attendu. Le borrow checker, quant à lui, vérifie la *sûreté mémoire* : le programme n'accède jamais à des variables non allouées ou non initialisées, et si deux pointeurs sont des alias (actifs en même temps et pouvant accéder à la même zone mémoire), alors aucun ne doit permettre une mutation de la mémoire.

Dans ce projet, nous ciblons un petit sous-ensemble du langage Rust, avec un ensemble de fonctionnalités limité, que nous appelons MiniRust. Nous fournissons le code du front-end : les analyseurs lexicaux et syntaxiques, le vérificateur de types, ainsi qu'une passe de traduction du langage de surface en MiniMir, un langage intermédiaire construit avec un graphe de flot de contrôle, inspiré de MIR, un langage intermédiaire utilisé par Rust. Nous fournissons également certains composants du borrow checker, ainsi que sa structure générale. **Votre tâche est de compléter les parties laissées incomplètes pour programmer un borrow checker fonctionnel pour notre langage jouet.** L'objectif est d'imiter autant que possible le comportement du compilateur Rust sur le sous-ensemble du langage choisi. Si certains comportements ne sont pas bien documentés dans ce fichier ou dans les commentaires du squelette, vous pouvez tester le comportement de Rust pour les reproduire.

La conception du borrow checker original de Rust est décrit sur une page web [1]. Même si cette documentation peut vous aider à comprendre le travail qui vous est demandé, gardez en tête que Rust est bien plus complexe que MiniRust – et son borrow checker également.

1.1 Le front-end de MiniRust

MiniRust est un petit sous-ensemble du langage Rust : les types incluent le type entier `i32`, les enregistrements déclarés par `struct`, les emprunts mutables et partagés, ainsi que le type `()`.

1. https://rustc-dev-guide.rust-lang.org/borrow_check.html

Les fonctions et les déclarations de type `struct` peuvent être polymorphes en durée de vie (lifetime), mais pas en type. Les traits ne sont pas supportés, mais un type d'enregistrement peut être rendu `Copy` si tous ses champs sont `Copy`, via l'annotation `#[derive(Copy, Clone)]`.

Les corps de fonctions contiennent des déclarations de variables introduites par `let`, toujours annotées explicitement par un type ; des structures de contrôle `loop`, `while`, `break`, `return`, `if` ; des opérations arithmétiques et booléennes simples ; des littéraux entiers suffixés par `i32` ; des accès et des constructions de structures et d'emprunts.

Il existe deux syntaxes pour écrire les types d'emprunt en MiniRust. Les prototypes de fonctions et déclarations de type `struct` utilisent des types complètement annotés : chaque emprunt est explicitement annoté par une durée de vie. En revanche, les types dans les déclarations `let` sont *effacés* : ils ne mentionnent jamais de durée de vie, qui doivent donc être inférées par le borrow checker.

Le vérificateur de types de MiniRust est bien plus simple que celui de Rust : certaines de ses importantes fonctionnalités ne sont pas implémentées. MiniRust ne fait pas d'« *autoderef* » : si `x` est de type `&S` (où `S` est un `struct` avec un champ `f`), MiniRust ne permet pas d'écrire directement `x.f` au lieu de `(*x).f`. De plus, MiniRust ne fait pas de réemprunt implicite : par exemple, si `x: &mut i32` et `f` est de prototype `fn f<'a>(a: &'a mut i32)`, alors le code `f(x); f(x)` est interdit en MiniRust, car la première utilisation de `x` consomme sa propriété. Ce code est cependant autorisé en Rust, car il est implicitement traduit en `f(&mut *x); f(&mut *x)`. Même si ce n'est pas une différence du borrow checker à proprement parler, cette fonctionnalité manquante peut expliquer certaines divergences observées entre les implémentations. Notez cependant que les tests fournis sont soigneusement conçus pour éviter ce cas.

1.2 Conception du borrow checker

Le borrow checker de MiniRust suit la conception de celui de Rust. Il ne travaille pas sur le langage de surface, mais sur le code MiniMir, un langage interne construit avec des graphes de flot de contrôle. Le code MiniMir est généré par la fonction `emit_fun` dans `emit_minimir.ml`, déjà écrite et correctement appelée.

Il est important de comprendre que le borrow checker ne travaille pas sur des variables mais sur des *places* : des zones mémoire accessibles à partir d'une variable locale via une suite d'accès de champ ou de déréférencements. Si `x` est une variable, alors `x.f`, `*x.f`, `*x`, `(**(*x).f).g` sont des places (si le type le permet). Cela signifie que les variables locales peuvent être partiellement initialisées (par exemple, `x.f` est initialisé, mais pas `x.g`), ou partiellement empruntées, ou empruntées avec des durées de vie hétérogènes.

Le borrow checker de MiniRust effectue les opérations suivantes, certaines devant être codées par vous :

— Il détermine quelles places sont initialisées, et quand, via une analyse statique. Il émet une erreur si une place est utilisée alors qu'elle est non initialisée. Le mot « initialisé » est à prendre au sens large : lorsqu'une place contient une valeur non `Copy` et que son contenu est déplacé, la place est considérée comme non initialisée. **Votre première tâche** est de compléter cette analyse dans `uninitialized_places.ml`. Ensuite, le fichier `borrowck.ml` effectuera les vérifications nécessaires à partir de cette analyse.

— Le borrow checker vérifie qu'aucune écriture n'a lieu sous un emprunt partagé, et qu'aucun emprunt mutable n'est créé avec une place se trouvant sous un emprunt partagé. **C'est votre seconde tâche**.

— Il effectue une analyse de vivacité (*liveness analysis*) pour déterminer, pour chaque variable locale et pour chaque point de programme, si la variable peut être utilisée dans le futur. Cette analyse est déjà programmée, et vous n'avez rien à faire.

— Il effectue une inférence des durées de vie (fonction `compute_lft_sets`) : puisque les types des variables locales sont effacés (les durées de vie sont omises), l'analyseur doit les déduire automatiquement. Lors de la génération du code MiniMir, la fonction `Emit_minimir.emit_fun` prépare ce travail en instanciant tous ces types avec des variables de durée de vie fraiches. L'analyseur d'emprunts génère ensuite des contraintes, qu'il résout. **Votre troisième tâche** consiste à compléter le code responsable de la génération de ces contraintes, situé dans `borrowck.ml`. Le code de résolution des contraintes (permettant de déterminer à quel point du programme chaque durée de vie doit être valide) est déjà fourni.

  Il existe deux types de contraintes. Tout d'abord, les *contraintes de survie* (ou « outlives ») indiquent qu'une durée de vie doit être plus longue qu'une autre. Ces contraintes sont générées par le typage du code MiniMir, en tenant compte des durées de vie (ce que l'analyseur de types principal de MiniRust ne faisait pas). Une attention particulière doit être portée aux réemprunts : lorsque le contenu d'un emprunt est à son tour emprunté, la durée de vie du réemprunt doit être strictement plus courte que celle de l'emprunt initial.

  Le second type de contraintes sont les *contraintes de vie* : elles indiquent qu'une durée de vie doit être valide à un point de programme donné. Les variables de durée de vie rigides (celles qui apparaissent comme paramètres génériques) doivent être valides en tout point du corps de la fonction. De plus, si une variable est vivante (au sens de l'analyse de vivacité) à un point donné, alors toute durée de vie apparaissant dans son type doit être considérée comme vivante à ce point.

  Après l'exécution du solveur, chaque durée de vie est associée à un ensemble de points du programme, correspondant aux endroits où elle doit être valide. Certains éléments spéciaux, de la forme `PpInCaller lft`, correspondent à des points du programme de la *fonction appelante*, où la durée de vie `lft`, un paramètre générique de la fonction courante, meurt. Ainsi, si `PpInCaller lft` fait partie de l'ensemble des points associés à une durée de vie `lft'`, cela signifie que `lft'` doit être valide lorsque `lft` meurt : elle doit donc vivre plus longtemps que `lft`.

  Si `lft` et `lft'` sont toutes deux des durées de vie génériques, alors la contrainte de survie correspondante doit être explicitement déclarée dans le prototype de la fonction. **Votre quatrième tâche** consiste à effectuer cette vérification.
— L'analyseur d'emprunts utilise les ensembles de durées de vie calculés précédemment pour déterminer quel emprunt est actif à quel point de programme, et donc quelles places sont encore empruntées à un point donné du programme. (Certaines opérations sont interdites sur une place empruntée : par exemple, l'écriture est interdite et la lecture l'est aussi si l'emprunt en question est mutable.) Ceci est le résultat d'une troisième analyse statique, implémentée dans `active_borrows.ml`. Elle vous est fournie : vous n'avez rien à faire à ce sujet.
— L'analyseur d'emprunts utilise le résultat de l'analyse des emprunts actifs décrite ci-dessus pour émettre une erreur s'il existe un emprunt actif en conflit avec certaines opérations effectuées par le code MiniMir. Une partie de ce code est déjà écrite, mais **votre cinquième tâche** consiste à le compléter.

2 Détails pratiques

2.1 Installation des dépendances logicielles

Pour compiler MiniRust, vous aurez besoin d'OCaml 5.1.1 (ou plus récent), de Menhir, de la bibliothèque Fix et de OCamlformat. Pour installer ces dépendances, utilisez Opam. Nous vous

suggérons de créer un *switch* Opam local[2] :

```
$ opam switch create . --deps-only --with-doc --with-test
$ eval $(opam env)
```

Vous aurez aussi besoin d'un compilateur Rust récent.

2.2 Développement

Une fois l'environnement prêt, vous pouvez compiler le projet avec :

```
$ dune build
```

Vous pouvez exécuter MiniRust sur un fichier d'entrée spécifique. La commande termine sans rien afficher si le fichier est accepté par minirust, et sinon affiche une erreur.

```
$ dune exec minirust/minirust.exe tests/01.rs
```

En définissant la variable d'environnement `MINIMIR`, l'exécution de MiniRust affichera sur la sortie standard le code MiniMir généré par le front-end :

```
$ MINIMIR=1 dune exec minirust/minirust.exe tests/01.rs
```

Vous pouvez aussi lancer toute la suite de tests :

```
$ dune runtest
```

Si vous souhaitez utiliser le formatage automatique de code, assurez-vous que votre éditeur de texte applique `OCamlformat` à chaque modification de fichier, de manière à suivre automatiquement le style de codage du projet. Si besoin, vous pouvez formater manuellement l'ensemble du code avec :

```
$ dune fmt
```

Notez que l'utilisation du formatage de code automatique n'est pas obligatoire, mais la lecture d'un code mal indenté sera difficile pour votre correcteur, qui pourra vous pénaliser pour cette raison.

2.3 Contenu de l'archive

Le dossier `tests` contient la suite de tests que nous vous fournissons pour vous guider dans le processus de développement. Vous pouvez en ajouter, mais ne modifiez pas ceux existants. Votre projet doit passer tous les tests : `dune runtest` ne doit rapporter aucune erreur.

Le dossier `minirust` contient le point d'entrée de MiniRust. Il ne contient qu'un petit morceau de code, qui appelle le code contenu dans le dossier `lib`. Celui-ci contient le code du vérificateur de types de MiniRust et du borrow checker :

- `error.{ml,mli}` définissent les fonctions d'affichage des erreurs ;
- `type.{ml,mli}` contiennent la définition du type des types de MiniRust, et quelques fonctions de manipulation des types et des durées de vie ;
- `lexer.{mll,mli}`, `parser.mly` et `parser_entry.{ml, mli}` contiennent les analyseurs lexical et syntaxique ;

2. https://opam.ocaml.org/blog/opam-local-switches/

— `ast.{ml,mli}` contiennent les définitions des types de l'arbre de syntaxe abstraite de MiniRust, ainsi que quelques fonctions facilitant leur manipulation ;
— `minimir.{ml,mli}` contiennent la définition du langage intermédiaire MiniMir ;
— `typecheck.{ml,mli}` et `emit_minimir.{ml,mli}` contiennent le vérificateur de types et la passe de traduction du langage de surface vers MiniMir ;
— `active_borrows.{ml,mli}`, `live_locals.{ml,mli}`, `uninitialized_places.{ml,mli}` contiennent les analyses statiques nécessaires au borrow checker ;
— `borrowck.{ml,mli}` contiennent le cœur du borrow checker.

2.4 Besoin d'aide ?

Discuter du projet avec d'autres étudiants est autorisé, mais chaque caractère de code présent dans votre archive finale doit avoir été écrit par vous-même, individuellement. Le plagiat sera sévèrement sanctionné.

Si certains aspects de ce sujet ne sont pas clairs pour vous, nous pouvons répondre à vos questions. Vous pouvez soit poser vos questions à la fin de l'un des cours ou TD, soit envoyer un e-mail à Jacques-Henri Jourdan.

3 Travail attendu et évaluation

— Avant de commencer à écrire la moindre ligne de code, vous devez prendre le temps de bien comprendre le projet. En plus de lire le présent document, **vous devez lire entièrement le contenu des fichiers** `active_borrows.ml*`, `borrowck.ml*`, `error.mli`, `live_locals.mli`, `minimir.mli`, `type.mli` et `uninitialized_places.ml*`. (Notez que pour certains modules, les fichiers `.ml` et `.mli` sont tous deux intéressants, tandis que pour d'autres, seul le fichier `.mli` mérite d'être lu.)
— Nous fournissons une implémentation de référence du projet, complète et disponible en ligne : https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/minirust/. Tout au long du projet, il est **fortement recommandé** de tester votre compréhension du sujet en **écrivant de nouveaux tests de votre invention** afin de comprendre le comportement attendu sur l'implémentation de référence, et valider votre implémentation. Pour ajouter un test, il suffit de l'écrire dans un fichier `.rs` dans le dossier `tests/`.
— Les fonctionnalités à implémenter sont dans les fichiers `uninitialized_places.ml` et `borrowck.ml`, indiquées par les commentaires `TODO`.
— Vous devez enfin rédiger un fichier `README.md` nous expliquant votre travail et les difficultés rencontrées.

Il est important de comprendre que les tests ne sont pas exhaustifs : il est nécessaire de **comprendre ce que vous faites** et d'écrire du code correct **pour toutes les entrées** afin d'obtenir une bonne note. Votre note sera déterminée après la lecture de votre code. Un code incorrect qui passe tous les tests n'obtiendra pas la note maximale.

La **correction** de votre code est essentielle ; ses performances ne le sont pas. Il est tout à fait acceptable de privilégier la clarté et la concision à l'efficacité. De plus, la **lisibilité** du code sera prise en compte : par exemple, un code mal indenté, avec des lignes trop longues, ou des noms de variables mal choisis pourra être pénalisé.

4 Extensions

Quelques points de la note finale sont réservés pour une extension éventuelle que vous ajouterez à votre projet. Pour les obtenir, ou simplement pour le plaisir, vous pouvez aller au-delà de ce qui est strictement demandé jusqu'ici. Ce que vous choisissez de faire dépend entièrement de vous. Voici quelques suggestions d'extensions possibles. Attention : nous ne savons pas exactement à quel point ces extensions sont difficiles ou chronophages.

— Ajouter la prise en charge du sous-typage des durées de vie (par exemple, `&'a i32` est un sous-type de `&'b i32` dès que `'a: 'b`). Comme en Rust, la variance des emprunts et des types `struct` doit être prise en compte.
— Étendre le système de types : ajouter la prise en charge des types `enum`, `Box`, du polymorphisme de types, etc.
— Développer un back-end qui traduit le code MiniMir en un langage bas-niveau (assembleur, Wasm, C, ...).

5 À envoyer

Envoyez à Jacques-Henri Jourdan une archive `.tar.gz` contenant votre projet ou un lien vers un dépôt Git le contenant. Si vous envoyez une archive, nous vous demandons d'y **inclure le dossier `.git` contenant votre historique de développement**. Nous ne consulterons pas le détail de cet historique, mais nous apprécierons la progression de votre travail dans le temps.

Afin de réduire la taille de l'archive, merci de ne **pas** inclure les fichiers générés automatiquement : `_build`, `_opam`, etc.

Veuillez inclure un fichier **README.md** pour décrire ce que vous avez réalisé. Vous êtes encouragé à fournir des explications sur votre solution ou les difficultés que vous avez rencontrées.

Attendez un accusé de réception dans les jours suivants.

6 Date de rendu

Envoyez votre projet avant le **dimanche 25 mai 2025**, à 23h59.