MPRI 2.4
Programmation fonctionnelle et systèmes de types
Programming project

Jacques-Henri Jourdan

2024–2025

1 Overview

The purpose of this project is to implement a "borrow checker" for an idealized subset of the Rust programming language. The borrow checker is an important piece of the Rust compiler: together with the type checker, these two components guarantee the safety of the Rust language. If a program is accepted by the compiler (both the type checker and the borrow checker validate it), then it is guaranteed that it will not have any undefined behavior.

In Rust, the type checker guarantees *type safety*: the values that the program manipulates always have the expected type. The borrow checker, on the other hand, checks *memory safety*: the program never accesses unallocated or uninitialized variables, and, if two pointer alias (they are both active at the same time, and there exists a memory location they can both access), then none of them can allow mutation of memory.

In this project, we target a toy subset of the Rust programming language, with a limited set of features, which we call MiniRust. We provide the front-end: the lexer, the parser, the type checker, and a pass which translates the surface language into MiniMir, a CFG-based internal language inspired by MIR, an intermediate language used by Rust. We also provide some components of the borrow checker, and its general structure. Your task is to fill the holes that we left in the borrow checker in order to program a functioning borrow checker for our toy language. The goal is to mimic as much as possible the behavior of the Rust compiler on the chosen language subset: if some behavior is not well documented in this document and in the comments of the template, you may test Rust's behavior to reproduce it.

The design of the original borrow checker of Rust is described in a web page.[1] While this piece of documentation can be helpful to understand the work you are assigned in this project, it is important to remember that Rust is significantly more complex than MiniRust, and so is its borrow checker.

1.1 MiniRust's front-end

MiniRust is a small subset of the Rust programming language: in MiniRust, types include the integer type `i32`, records declared by `struct`, mutable and shared borrows, and the unit type.

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

Both functions and `struct` declarations can be lifetime polymorphic, but not type polymorphic. Type traits are not supported, but a record type can be made `Copy` when all of its fields have `Copy` types by adding `#[derive(Copy, Clone)]` before its declaration.

Function bodies contain let bindings for one variable, always explicitly annotated with a type; `loop`, `while`, `break`, `return` and `if` control constructs; simple arithmetic and Boolean operations; integer literals suffixed with `i32`; access and construction of records and borrows.

There are two syntaxes to write borrow types in MiniRust. Function prototypes and `struct` declarations contain fully annotated types: every borrow is explicitly annotated with a lifetime. On the other hand, types annotating let bindings are *erased*: they never mention lifetimes, which should be inferred by the borrow checker.

The type checker of MiniRust is much simpler than that of Rust: we did not implement some of its salient features. MiniRust does not have "autoderef": if `x` has type `&S`, where `S` is a `struct` type with a field `f`, then, contrarily to Rust, MiniRust does not allow accessing `(*x).f` directly with `x.f`. In addition, MiniRust does not implement implicit reborrowing: for example, if `x: &mut i32` and `f` is a function of prototype `fn f<'a>(a: &'a mut i32)`, then code `f(x); f(x)` is forbidden in MiniRust, because the first use of `x` consumes its ownership. This code is however allowed in Rust, because it is implicitly translated into `f(&mut *x); f(&mut *x)`. Even though this is not a difference in the borrow checker, this missing feature can explain a lot of the differences one can observe when comparing the behaviors of both borrow checkers. Note, however, that the tests we provide are carefully designed to avoid this issue.

1.2 The design of the borrow checker

MiniRust's borrow checker follows the design or Rust's. It does not work on the surface language, but instead handles code in MiniMir, an internal language based on a control flow graph. MiniMir code is generated by the function `emit_fun` in `emit_minimir.ml`. This function is already written, and properly called.

Importantly, the borrow checker works at the level of *places*: these are regions of memory accessible from a local variable via a sequence of field accesses or borrow dereferences. If `x` is a variable, then `x.f`, `*x.f`, `*x`, `(**(*x).f).g` are places (assuming `x` has an appropriate type). This means that local variables can be partially initialized (i.e., `x.f` is initialized, but not `x.g`) or partially borrowed, or borrowed for heterogeneous lifetimes.

MiniRust's borrow checker performs the following operations, some of which you need to program:

- By using a dataflow analysis, it determines which places are initialized, and when. It emits an error if a place is used while uninitialized. The word "initializedness" has to be taken in a broad meaning: when a variable contains a value which is not Copy, and its content is consumed by moving its resources, then this variable is considered uninitialized, even though it has been initialized earlier in the code. Your first task is to complete the implementation of this dataflow analysis in `uninitialized_places.ml`. Then, the code in `borrowck.ml` will perform the appropriate checks on MiniMir code, depending on the result of your analysis.

- The borrow checker makes sure that no write is performed below a shared borrow, and that no mutable borrow is taken below a shared borrow. Your second task is to write this code.

- The borrow checker does a liveness analysis to determine, for each local variable and program

point, whether the value of the local variable may be used in the future. This analysis is already programmed, you do not need to do anything.

- It does lifetime inference (function `compute_lft_sets`): because the types of the local variables are erased (lifetime are omitted), the borrow checker has to guess what they correspond to. When generating MiniMir code, `Emit_minimir.emit_fun` prepares this work by instantiating all these types with fresh lifetime variables. The borrow checker generates constraints, which it solves. Your third task is to complete the code generating these constraints in `borrowck.ml`. The code solving these constraints to determine when each lifetime need to be alive at which program point is given.

  There are two kinds of constraints: first, outlives constraints state that some lifetime needs to be longer than another. These constraints are generated by type-checking MiniMir code, taking into account lifetime information (which the main type checker of MiniRust ignored). Special care should be taken for reborrows: when the content of a borrow is reborrowed, the reborrow lifetime should be made shorter than the lifetime of the original borrow.

  The second kinds of constraints are living constraints: they state that some lifetime needs to be alive at some program point. Rigid lifetime variables (appearing as generic parameters) should be made alive at any program point in the function. In addition, if a variable is live (in the sense of the liveness analysis) at some program point, then any lifetime appearing free in it type should be made alive at this program point.

  After the solver has run, each lifetime is associated with a set of program points, corresponding to the program points where that lifetime needs to be alive. Some special elements of the form `PpInCaller lft` correspond to program points of the *calling function*, where the lifetime `lft`, a generic parameter of the current function, dies. Hence, if `PpInCaller lft` is in the set of program points associated with lifetime `lft'`, this means that `lft'` should be alive when `lft` dies: it needs to outlive `lft`. If both `lft` and `lft'` are generic lifetimes, then the outlives constraint need to be declared in the prototype of the function. Your fourth task is to do this check.

- The borrow checker uses the lifetime sets computed above to determine which borrow is active at which program location, and therefore which place are still borrowed at a given program location. (Some operations are forbidden on a borrowed place: for example, writing is forbidden, and reading is forbidden if the borrow in question is mutable.) This is the result of a third dataflow analysis, programmed in `active_borrows.ml`. This is provided to you: you do not need to do anything.

- The borrow checker use the result of the active borrow analysis above to emit an error if there exists an active borrow conflicting with some operations performed by the MiniMir code. Some of this code is already written, but your fifth task is to complete it.

2 Practical details

2.1 Install software dependencies

To build MiniRust, you will need OCaml 5.1.1 or newer (an older version may work, without any guarantee), Menhir, the Fix library, and OCamlformat. In order to install these OCaml dependen-

cies, install Opam,[2] the OCaml package manager, on your system. If you have never used Opam before, you need to initialize it (otherwise, skip this step):

```
$ opam init
```

For convenience, we suggest setting up a local Opam distribution[3] using the following commands:

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

In addition, a relatively recent Rust compiler is needed to run the tests. If your distribution provides a Rust compiler, then it is probably good enough. Otherwise, follow the instructions on the Rust web site.[4]

2.2 Development

Once your programming environment is setup, you can build the project with:

```
$ dune build
```

You can execute MiniRust on a specific input file using:

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

You can also run all the test suite with:

```
$ dune runtest
```

If you want to use automatic formatting, make sure that your text editor applies OCamlformat each time a file is modified, so as to automatically follow the project's coding style. If need be, you can manually format the whole codebase with:

```
$ dune fmt
```

2.3 Contents of the archive

The directory `tests` contains the test suite we provide to guide you in the development process. You are allowed to add some tests, but you should not to change any of the existing ones. A correctly completed project should pass all the tests: `dune runtest` should not report any error.

The directory `minirust` contains the code of the MiniRust type checker and borrow checker:

- `minirust.ml` is the main entry point of the program;
- `error.ml` and `error.mli` define the error reporting function (which you should use);
- `ast_types.mli` contains the definitions of the abstract syntax tree types of MiniRust;
- `type.ml` and `type.mli` contain the definition of the type of MiniRust types, and some type and lifetime manipulating functions;

[2] https://opam.ocaml.org/doc/Install.html
[3] https://opam.ocaml.org/blog/opam-local-switches/
[4] https://www.rust-lang.org/tools/install

- `lexer.mll`, `lexer.mli` and `parser.mly` contain the lexer and parser of MiniRust;
- `ast.ml` and `ast.mli` contain the parsed AST of the input file, together with some accessors;
- `minimir.ml` and `minimir.mli` contain the definition of the MiniMir internal language;
- `typecheck.ml`, `typecheck.mli`, `emit_minimir.ml` and `emit_minimir.mli` contain the type checker and the translation of the surface language to MiniMir;
- `active_borrows.{ml,mli}`, `live_locals.{ml,mli}`, and `uninitialized_places.{ml,mli}` contain the dataflow analyses that the borrow checker uses to validate the program (or declare it invalid);
- `borrowck.ml` and `borrowck.mli` define the main entry point of the borrow checker.

2.4 Need help?

Discussing with other students about the project is allowed, but any single character of code you send in your final archive should be written by you individually. Plagiarism will be severely sanctioned.

If you cannot understand some aspects of this subjection, we can answer some of your questions. You can either ask Jacques-Henri Jourdan at the end of one of his courses, or send him an e-mail.

3 Expected work and evaluation

Before starting to write any piece of code, you should take some time to understand the project. To that end, in addition to reading the present document, you should read entirely the contents of the files `active_borrows.ml*`, `borrowck.ml*`, `error.mli`, `live_locals.mli`, `minimir.mli`, `type.mli` and `uninitialized_places.ml*`. (Note that for some modules, both the `.ml` and `.mli` are interesting, while for others, only the `.mli` is worth reading.)

Then, you should complete the files `uninitialized_places.ml` and `borrowck.ml` by addressing the `TODO` comments. After having written a first version of your code, you should run the tests, and address the ones which are still failing.

Finally, you should write a `README.md` file explaining your work and the difficulties you encountered.

Assignments will be evaluated by a combination of:

- **Testing**. Your compiler will be tested with the input programs that we provide (make sure that `dune runtest` succeeds!) and with additional input programs.
- **Reading**. We will browse through your source code and evaluate its correctness and elegance.

The **correctness** of your code matters; its performance does not. It is acceptable to favor clarity and conciseness over efficiency.

4 Extra credit

For extra credit, or just for fun, you may wish to go beyond what is strictly requested. What to do is up to you. Here are some suggestions of things to do. Beware: we do not know exactly how difficult or time-consuming these extensions are.

- Write additional tests that would reflect some particularly subtle behavior of the borrow checker that would not be tested by the proposed tests.
- Add support for lifetime subtyping (e.g., `&'a i32` is a subtype of `&'b i32`, as soon as `'a: 'b`). As in Rust, variance of borrows and `struct` types should be taken into account.
- Extend the type system: add support for `enum` types, `Box`, type polymorphism...
- Develop a back-end that translates MiniMir code into a low-level language (Assembly, Wasm, C, ...).

5 What to send

When you are done, please send to Jacques-Henri Jourdan a `.tar.gz` archive containing your completed programming project. The archive should contain a single directory `mpri-2.4-projet-2024-2025`. Expect an acknowledgement within a few days.

Do **not** include any automatically generated files such as build artifacts, version control directories, or package manager directories. In particular, any `_build`, `.git` or `_opam` directory should not be included. If you include these, your archive may be rejected by the e-mail system because it is too large or contain executable files.

Please include a file **README.md** to describe what you have achieved. You are welcome to provide explanations (in French or in English) about your solution or about the difficulties that you have encountered.

6 Deadline

Please send your project before **Friday, February 28, 2025**, at 23:59.