Freie Universität Berlin

Implementierung und Evaluation von Vorbedingungskommunikation für `unsafe` Code in Rust

Zwischenpräsentation der Bachelorarbeit

Niclas Schwarzlose (niclas.schwarzlose@fu-berlin.de)

06.07.2020

Inhalt

- Problembeschreibung
- Lösungsansatz
- Umsetzung
- Status
- Evaluation
- Ausblick
- Verwandte Arbeiten

Rust Übersicht

Rust ist eine Programmiersprache mit Fokus auf

- Performanz
  - (u.a. keine Garbage Collection)
- Zuverlässigkeit
  - (u.a. Speichersicherheit)
- Produktivität

Warum existiert `unsafe` Code?

Die Ziele von Rust werden durch bestimmte Regeln garantiert.

Es gibt jedoch Situationen, in denen die Regeln zu einschränkend sind.

In diesen Situationen können die Regeln mit `unsafe` Code umgangen werden.

Warum heißt es `unsafe` Code?

> *[`unsafe` code is used] to declare the existence of contracts the compiler can't check, and to declare that a programmer has checked that these contracts have been upheld.*

https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html

- Nicht-Einhaltung dieser Verträge führt zu undefiniertem Verhalten und damit zu möglichen Abstürzen/Sicherheitslücken.
- Die Verantwortung dafür liegt bei den Programmierer*innen.

Was kann `unsafe` Code?

- Zeiger dereferenzieren
- **`unsafe` Funktionen aufrufen**
- `unsafe` Merkmale (traits) implementieren
- Veränderbare `static` Variablen verwenden
- Auf `union`-Felder zugreifen

https://doc.rust-lang.org/nomicon/what-unsafe-does.html

Wie macht man `unsafe` Funktionen "safe"?

- Man umgibt sie mit einem `unsafe` Block.
- Man dokumentiert, was beim Aufruf beachtet werden muss.
- Man beachtet die Verträge (Vorbedingungen) beim Aufruf der Funktion.

  Idealerweise wird hier noch in einem Kommentar dokumentiert, warum man glaubt, dass die Vorbedingungen eingehalten werden.

```
error[E0133]: call to unsafe function is unsafe and requires unsafe function or block
 --> src/main.rs:4:5
  |
4 |     unsafe_fn();
  |     ^^^^^^^^^^^ call to unsafe function
  |
  = note: consult the function's documentation for information on how to avoid undefined behavior
```

Was sind mögliche Probleme dabei?

Ein*e Programmierer*in von einer `unsafe` Funktion kann

- vergessen die Vorbedingungen zu dokumentieren.

Ein*e Programmierer*in vom Aufruf einer `unsafe` Funktion kann

- vergessen in die Dokumentation zu schauen und die Vorbedingungen nicht beachten.
- eine oder mehrere Vorbedingungen in der Dokumentation übersehen.
- einen Fehler bei der Überprüfung der Vorbedingungen machen.
- nach einem Update veränderte Vorbedingungen für die Funktion nicht bemerken.
- nicht dokumentieren, warum die Vorbedingungen eingehalten werden, was es Nachfolgern, die den Code ändern müssen, leichter macht Fehler zu machen.

Der Lösungsansatz

- Vorbedingungen können an Funktionsdefinitionen in maschienenlesbarer Form vorgehalten werden.
- Programmierer*innen müssen beim Aufruf der Funktionen angeben, dass die Vorbedingungen beachtet wurden und warum sie gelten.
- Der Kompiler soll überprüfen, ob die Vorbedingungen bei der Definition und beim Aufruf übereinstimmen.
- Löst einige, aber nicht alle der genannten Probleme.

Beispiel

```rust
use pre::pre;

#[pre("`arg` is a meaningful value")]
fn foo(arg: i32) {
  assert!(arg == 42);
}

#[pre]
fn main() {
  #[assure(
    "`arg` is a meaningful value",
    reason = "42 is meaningful"
  )]
  foo(42);
}
```

Wie geht man mit Fremdcode um?

```rust
#[pre::extern_crate(std)]
mod pre_std {
  mod mem {
    #[pre("an all-zero byte-pattern is valid for `T`")]
    unsafe fn zeroed<T>() -> T;
  }
}

#[pre::pre]
fn main() {
  #[assure(
    "an all-zero byte-pattern is valid for `T`",
    reason = "`i32` supports an all-zero byte-pattern"
  )]
  let x: i32 = unsafe { pre_std::mem::zeroed() };
}
```

Verschiedene Vorbedingungstypen

```rust
use pre::pre;

#[pre(valid_ptr(pointer, r))]
fn read(pointer: *const i32) -> i32 {
  *pointer
}

#[pre(0 <= val && val < 5)]
fn do_stuff(val: i32) -> i32 {
  [0, 1, 2, 3, 4][val as usize]
}
```

Umsetzung der Funktionsdefinition (nightly)

```rust
#[pre::pre("some_val > 42.0")]
fn has_preconditions(some_val: f32) {
  /* ... */
}
```

wird zu

```rust
fn has_preconditions(
  some_val: f32,
  _: ::core::marker::PhantomData<(
    ::pre::CustomConditionHolds<"some_val > 42.0">,
  )>,
) {
  /* ... */
}
```

Umsetzung des Aufrufs (nightly)


```rust
#[pre::pre]
fn main() {
  #[assure("some_val > 42.0", reason = "43.0 > 42.0")]
  has_preconditions(43.0);
}
```

wird zu

```rust
fn main() {
  has_preconditions(
    43.0,
    ::core::marker::PhantomData::<(
      ::pre::CustomConditionHolds<"some_val > 42.0">,
    )>,
  );
}
```

Umsetzung der Funktionsdefinition (stable)

```rust
#[pre::pre("some_val > 42.0")]
fn has_preconditions(some_val: f32) {
  /* ... */
}
```

wird zu

```rust
struct has_preconditions {
  _custom_some__val_20_3e_2042_2e0: (),
}

fn has_preconditions(
  some_val: f32,
  _: has_preconditions
) {
  /* ... */
}
```

Umsetzung des Aufrufs (stable)

```rust
#[pre::pre]
fn main() {
  #[assure("some_val > 42.0", reason = "43.0 > 42.0")]
  has_preconditions(43.0);
}
```

wird zu

```rust
fn main() {
  has_preconditions(
    43.0,
    has_preconditions {
      _custom_some__val_20_3e_2042_2e0: (),
    },
  );
}
```

Umsetzung (Fremdcode)

```
#[pre::extern_crate(std)]
mod pre_std {
  mod ptr {
    #[pre(valid_ptr(dst, w))]
    unsafe fn write_unaligned<T>(dst: *mut T, src: T) -> T;
  }
}
```

wird zu

```
pub mod pre_std {
  pub use std::*;
  pub mod ptr {
    pub use std::ptr::*;
    #[pre(valid_ptr(dst, w))]
    pub unsafe fn write_unaligned<T>(dst: *mut T, src: T) -> T {
      std::ptr::write_unaligned(dst, src)
    }
  }
}
```

Umsetzung der Funktionsdefinition (Fremdcode Methoden)

```rust
#[pre::extern_crate(std)]
pub mod pre_std {
  mod vec {
    impl<T> Vec<T> {
      #[pre("...")]
      unsafe fn set_len(&mut self, new_len: usize);
    }
  }
}
```

wird zu

```rust
pub mod pre_std {
  pub mod vec {
    #[pre("...")]
    pub fn Vec__impl__set_len__() {}
  }
}
```

Umsetzung des Aufrufs (Fremdcode Methoden)

```rust
#[pre::pre]
fn main() {
  let mut v = vec![true, false, true];
  unsafe {
    #[forward(impl pre_std::vec::Vec)]
    #[assure("...", reason = "...")]
    v.set_len(0);
  }
}
```

wird zu

```rust
#[pre::pre]
fn main() {
    let mut v = vec![true, false, true];
    unsafe {
        if true {
            v.set_len(0)
        } else {
            #[assure("...", reason = "...")]
            pre_std::vec::Vec__impl__set_len__();
            unreachable!()
        };
    }
}
```

Status

- Fast alle geplanten Features sind implementiert.
- Die Dokumentation der Bibliothek deckt bereits die wichtigsten Punkte ab.
- Es fehlen vor allem noch Tests für manche Bereiche.
- Insgesamt hat die Bibliothek momentan ca. 3500 Zeilen Code und Dokumentation.

Evaluation

- Ist eine solche Bibliothek hilfreich?
  - Wenn ja, wann?
- Ist eine solche Bibliothek erwünscht?
  - Wenn nein, warum nicht?
- Rechtfertigt die gewonnene Sicherheit den Mehraufwand?

Evaluationsstrategien

- Die Bibliothek veröffentlichen und um Verwendung und Feedback bitten.
- Einen Pull Request bei einem Open Source Projekt öffnen.
- Eine Codedurchsicht mithilfe der Bibliothek bei einem Open Source Projekt durchführen.
- Einen schon behobenen Fehler erneut “suchen” und prüfen, ob er gefunden worden wäre.

Pull Request Anfragetext

Hi, I've developed a crate for my bachelor's thesis to make working with `unsafe` code a little bit safer. It works by annotating `unsafe` functions with the preconditions that need to be upheld for their safety. A call to an annotated function will fail to compile, unless it is assured at the call site that the preconditions hold. This does not incur any cost at runtime in a release build. Here is the link to the crate: <link>.

If you'd be willing to give it a try, I'd be happy to prepare a pull request to integrate its usage into <project name>. If not, I'd appreciate it if you could briefly let me know why you're not interested, as that will be valuable feedback for my thesis.

Ausblick

- Bessere Fehlermeldung bei fehlendem `assure`.

```
error[E0061]: this function takes 1 argument but 0 arguments were supplied
 --> src/main.rs:8:5
  |
4 | fn foo() {}
  | -------- defined here
...
8 |     foo();
  |     ^^^-- supplied 0 arguments
  |     |
  |     expected 1 argument
```

- Besserer Umgang mit Methoden aus Fremdcode.
- Weitere Formen von `unsafe` Code unterstützen (z.B. Merkmale (`traits`)).
- Vorbedingungen an Methoden im “stable”-Kompiler unterstützen.

Verwandte Arbeiten

- D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis,” in *IEEE Software*, vol. 19, no. 1, pp. 42-51, Jan.-Feb. 2002, doi: 10.1109/52.976940.
- R. Jung, H.H. Dang, J. Kang, and D. Dreyer, “Stacked borrows: an aliasing model for Rust,” in *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Article 41 (January 2020), doi: 10.1145/3371109
- Z. Huang, Y. Wang and J. Liu, “Detecting Unsafe Raw Pointer Dereferencing Behavior in Rust,” in *IEICE TRANSACTIONS on Information and Systems*, vol. E101-D, no. 8, pp. 2150-2153, 2018, doi: 10.1587/transinf.2018EDL8040
- J. Toman, S. Pernsteiner and E. Torlak, “Crust: A Bounded Verifier for Rust (N),” 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, 2015, pp. 75-80, doi: 10.1109/ASE.2015.77.

Ende

Fragen?

Feedback?