Freie Universität Berlin

Implementierung und Evaluation von pre, einem Werkzeug zum besseren Umgang mit Vorbedingungen von unsicherem Code in Rust

Bachelorarbeitsverteidigung

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

05.10.2020

Inhalt

- Problembeschreibung
- Lösungsansatz
- Technische Umsetzung
- Evaluation
- Ausblick
- Zusammenfassung

Rust Übersicht

Rust ist eine Programmiersprache mit Fokus auf

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

Warum existiert unsicherer 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 unsicherem Code umgangen werden.

Warum heißt es “unsicherer” 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.*

Aus dem Rustonomicon (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 Entwickler*innen.

Was kann unsicherer Code?

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

Aus dem Rustonomicon (https://doc.rust-lang.org/nomicon/what-unsafe-does.html)

Wie macht man unsichere Funktionen sicher?

```
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
```

- 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.

Was sind mögliche Probleme dabei?

Autor*innen von unsicheren Funktion können

- vergessen die Vorbedingungen zu dokumentieren.

Entwickler*innen können beim Aufruf einer unsicheren Funktionen

- einen Fehler bei der Überprüfung der Vorbedingungen machen.
- **vergessen in die Dokumentation zu schauen und die Vorbedingungen nicht beachten**.
- **eine oder mehrere Vorbedingungen in der Dokumentation übersehen**.
- **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.
- Entwickler*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 bei korrekter Verwendung die fettgedruckten Probleme.

Beispiel

```rust
use pre::pre;

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

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

Verschiedene Vorbedingungstypen

```rust
use pre::pre;

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

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

Technische Umsetzung

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

```rust
unsafe fn foo(
  arg: i32,
  _: (::pre::CustomCondition<
    "`arg` is a meaningful value"
  >,),
) {
  assert!(arg == 42);
}
```

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

```rust
fn main() {
  unsafe {
    foo(
      42,
      (::pre::CustomCondition::<
        "`arg` is a meaningful"
      >,)
    );
  }
}
```

Evaluation

- Wann ist pre hilfreich und wann nicht?
- Besteht Interesse in der Rust-community an pre?
- Rechtfertigt die gewonnene zusätzliche Sicherheit den Mehraufwand?
- Was ist noch verbesserungswürdig?

Evaluationsstrategien

- pre veröffentlichen und um Feedback bitten.
- Eine Codedurchsicht mithilfe von pre bei einem Open Source Projekt (secstr) durchführen.

Verwendung von pre in secstr

```rust
pub fn cmp<T: Sized + Copy>(us: &[T], them: &[T]) -> bool {
    if us.len() != them.len() {
        return false;
    }

    let mut result: u8 = 0;

    let ptr_us   = us.as_ptr()   as *mut u8;
    let ptr_them = them.as_ptr() as *mut u8;
    for i in 0 .. size_of(us) {
        unsafe {
            result |= *(ptr_us.offset(i as isize)) ^
                *(ptr_them.offset(i as isize));
        }
    }

    result == 0
}
```

Verwendung von pre in secstr

```rust
pub fn cmp<T: Sized + Copy>(us: &[T], them: &[T]) -> bool {
    if us.len() != them.len() {
        return false;
    }

    let mut result: u8 = 0;

    let ptr_us   = us.as_ptr()   as *mut u8;
    let ptr_them = them.as_ptr() as *mut u8;
    for i in 0 .. size_of(us) {
        unsafe {
            result |= *(ptr_us.offset(i as isize)) ^
                *(ptr_them.offset(i as isize));
        }
    }

    result == 0
}
```

Verwendung von pre in secstr

```
result |= *(ptr_us.offset(i as isize)) ^
    *(ptr_them.offset(i as isize));
```

ist äquivalent zu

```
let ptr_us = ptr_us.offset(i as isize);
let val_us = std::ptr::read(ptr_us);
let ptr_them = ptr_them.offset(i as isize);
let val_them = std::ptr::read(ptr_them);

result |= val_us ^ val_them;
```

Verwendung von pre in secstr

```
#[assure(/* ... */)]
#[assure(/* ... */)]
#[assure(/* ... */)]
let ptr_us = ptr_us.offset(i as isize);
#[assure(valid_ptr(src, r), reason = "...")]
#[assure(proper_align(src), reason = "...")]
#[assure(
  "`T` is `Copy` or the value at `*src` isn't used
    after this call", reason = "..."
)]
#[assure(
  "`src` points to a properly initialized value of
    type `T`", reason = "..."
)]
let val_us = std::ptr::read(ptr_us);
```

Füllbytes (padding)

```
#[repr(C)]
struct Foo {
    a: u8,
    b: u16
}
```

Speicherlayout von einem Array von zwei `Foo` `structs` an Adresse `p`:

p + 0	p + 1	p + 2	p + 3	p + 4	p + 5	p + 6	p + 7
a	( )	b		a	( )	b	

Also ist nicht garantiert, dass die folgende Zusicherung gilt:

```
assert!(
    cmp(
        &[Foo { a: 0x3f, b: 0x87ab }],
        &[Foo { a: 0x3f, b: 0x87ab }]
    )
);
```

Evaluation: Ergebnis

- Es war möglich einen Fehler zu finden.
- Es gab gutes Feedback zu pre (aber nicht viel Feedback).
- pre hat auch noch einige Dinge, die noch verbesserungswürdig sind:

```
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
```

Ausblick

- Weitere Formen von unsicherem Code unterstützen (z.B. unsichere Merkmale (`traits`)).
- Mehr Unterstützungen hinzufügen (z.B. optionale Fehlermeldungen für unsichere Funktionen, die keine dokumentierten Vorbedingungen haben).

Zusammenfassung

- Beim Entwickeln von unsicherem Code können auf verschiedene Arten Fehler entstehen.
- pre hilft dabei Fehler mancher Art zu verhindern.
- Mithilfe von pre ist es auch möglich Fehler in schon bestehendem Code zu finden.
- Es gibt noch einige mögliche Verbesserungen für pre.

Ende

Vielen Dank für Ihre Aufmerksamkeit!

Fragen?

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() };
}
```

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,
  _: (::pre::CustomCondition<"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,
    (::pre::CustomCondition<"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)

```
#[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

```
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!()
        };
    }
}
```