Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

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

Niclas Schwarzlose

Matrikelnummer: 4975420

niclas.schwarzlose@fu-berlin.de

Betreuer: Victor Brekenfeld

Eingereicht bei: Prof. Lutz Prechelt

Zweitgutachter: Prof. Dr.-Ing. Jochen Schiller

Berlin, 19. August 2020

Zusammenfassung

Hintergrund Die Programmiersprache Rust legt einen Fokus auf durch den Compiler geprüfte Sicherheit. Für Situationen, in denen der Compiler eine fehlerfreie Lösung nicht als sicher beweisen kann, gibt es unsicheren Code, um die Einschränkungen des Compilers zu umgehen. Entwickler sind bei der Verwendung von unsicherem Code selbst dafür verantwortlich, dass ihre Software kein undefiniertes Verhalten haben. Um dies zu garantieren, müssen beim Aufruf von unsicheren Funktionen bestimmte Vorbedingungen eingehalten werden. Diese werden in der Regel jedoch nur in der Dokumentation festgehalten und sind damit völlig unabhängig vom Code.

Ziele In dieser Arbeit wurde *pre* entwickelt, ein Werkzeug, welches den Umgang mit Vorbedingungen von unsicheren Funktionen sicherer machen soll. *pre* funktioniert, indem Entwickler die Vorbedingungen von unsicheren Funktionen an der Funktionsdefinition annotieren. Beim Aufruf dieser Funktionen muss daraufhin begründet werden, warum die Vorbedingungen eingehalten werden. Wird dies nicht getan, führt das zu einem Kompilierungsfehler. Damit sind die Vorbedingungen ein fester Teil der Funktionssignatur und nicht mehr unabhängig vom Code.

Methoden Für die Implementierung von *pre* wurden prozedurale Attribut-Makros verwendet. Diese ermöglichen das Annotieren der Vorbedingungen und der Begründungen an Funktionsdefinitionen und -aufrufen. Die Attribut-Makros codieren die Vorbedingungen in der Funktionssignatur und in dem Funktionsaufruf.

Ergebnisse Um zu evaluieren, wie hilfreich *pre* in der Praxis ist, wurde *pre* in ein bestehendes Projekt, welches unsicheren Code verwendet, integriert. Hierbei wurde mithilfe von *pre* ein Fehler in diesem Projekt gefunden und behoben. Des Weiteren wurden Rust-Entwickler um Feedback zu *pre* gebeten. Aufgrund der positiven Rückmeldungen scheint es, als gäbe es unter Rust-Entwicklern ein Interesse an einem Projekt wie *pre*. Es gibt jedoch auch noch einige Beschränkungen, die die Verwendung von *pre* erschweren.

Schlussfolgerungen Ein Werkzeug wie *pre* hat das Potenzial, Entwickler bei der Verwendung von unsicherem Code zu unterstützen und Fehler zu vermeiden. *pre* ist jedoch noch nicht ausgereift genug, um als allgemeingültiges Werkzeug effektiv für diesen Zweck verwendet werden zu können.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

19. August 2020

Niclas Schwarzlose

Inhaltsverzeichnis

1 Einführung

Eine der wichtigsten Eigenschaften von Software ist Korrektheit. Nutzer wollen sich auf die Software, die sie verwenden, verlassen können. Fehler in Software können auf vielen Ebenen entstehen – von der Spezifikation der Software bis zur Maschienencodegenerierung im Compiler. Zur Vermeidung dieser Fehler gibt es verschiedene Werkzeuge, die unter anderem ganze Fehlerklassen unmöglich machen können. Die Programmiersprache Rust schließt mit ihrem Design beispielsweise das Entstehen von Speicherzugriffsfehlern aus [1][1], obwohl sie eine kompilierte Systemsprache ohne Garbage Collection ist. Um dies zu erreichen wird ein System von Eigentümerschaft und Ausleihen[2] verwendet. Während dieses System meist ausdrucksstark genug ist, gibt es doch manche Fälle – beispielsweise das Implementieren einiger Datenstrukturen – in denen es zu restriktiv ist. Für diese Fälle gibt es in Rust unsicheren[3] Code.

Unsicherer Code erlaubt Entwicklern auf unterschiedliche Arten und Weisen das Eigentümerschaftssystem zu umgehen. Im Gegenzug sind Entwickler selbst dafür verantwortlich, dass sie keine Speicherzugriffsfehler und kein undefiniertes Verhalten in ihre Software einbringen. Um das zu verhindern, müssen sie eine Reihe von Verträgen einhalten [2]. Diese Arbeit wird sich hauptsächlich mit unsicheren Funktionen beschäftigen, einem Teilaspekt von unsicherem Code[4]. Deswegen werden die Verträge hier im Weiteren als Vorbedingungen bezeichnet, da sie beim Aufruf der Funktion gelten müssen. Sie werden in der Regel in der Dokumentation der Funktion festgehalten[5]. Beim Aufruf von unsicheren Funktionen wird von Entwicklern erwartet, dass sie in die Dokumentation schauen und die Vorbedingungen einhalten. Bei Befolgung dieses Ansatzes können unter anderem folgende Situationen Probleme verursachen:

1. Die Autoren einer Funktion haben keine Vorbedingungen dokumentiert.

2. Die Entwickler vergessen beim Aufruf in die Dokumentation zu schauen. Dies kann insbesondere passieren, wenn der Aufruf sich schon aus einem anderen Grund in einem unsicheren Kontext[6] befindet.

3. Die Entwickler übersehen eine oder mehrere Vorbedingungen in der Dokumentation. Dies kann insbesondere bei längeren Texten, in denen mehrere Vorbedingungen beschrieben werden, der Fall sein.

4. Eine neue Version einer Bibliothek verändert die Vorbedingungen einer Funktion und den Nutzern fällt dies nicht auf.

5. Beim Aufruf einer unsicheren Funktion wurde nicht dokumentiert, warum die Vorbedingungen eingehalten werden. Bei einer späteren Änderung des Aufrufs

[1] Solange kein unsicherer Code verwendet wird

[2] Originalbegriffe: „ownership“ und „borrowing“

[3] Originalbegriff: „unsafe“

[4] Dies ist die am häufigsten verwendete Form von unsicherem Code. Da der Umfang einer Bachelorarbeit begrenzt ist, habe ich mich nur auf diesen Teilaspekt beschränkt.

[5] Dies spiegelt sich auch in der Fehlermeldung des Compilers wieder, wenn eine unsichere Funktion aus einem sicheren Kontext aufgerufen wird: `note: consult the function's documentation for information on how to avoid undefined behavior` (Compilerversion: 1.45.0)

[6] Im Inneren einer unsicheren Funktion oder in einem `unsafe` Block

wurde nicht erneut in die Dokumentation geschaut und deshalb ein Fehler eingebaut.

6. Die Entwickler nehmen fehlerhaft an, dass alle Vorbedingungen eingehalten werden.

In dieser Arbeit wurde *pre* [3] entwickelt, ein Werkzeug in Form einer Rust-Bibliothek, welches Lösungen für Situationen 2, 3, 4 und 5 bieten soll. *pre* erlaubt Autoren von unsicheren Funktionen die Vorbedingungen einer Funktion an ihrer Definition zu annotieren. Die Vorbedingungen werden daraufhin in die Funktionssignatur aufgenommen. Um die Funktion aufrufen zu können, muss die korrekte Signatur verwendet werden. Entwickler müssen sich also beim Aufruf von unsicheren Funktionen mit den Vorbedingungen auseinandersetzen, da sie sonst eine falsche Funktionssignatur verwenden. Damit soll verhindert werden, dass Vorbedingungen vergessen oder übersehen werden können. Außerdem führt dadurch eine Änderung der Vorbedingungen zu Fehlern beim Kompilieren statt zu potenziell unerkanntem undefinierten Verhalten. *pre* führt jedoch keine semantische Überprüfung durch. Es wird lediglich überprüft, dass das Einhalten aller Vorbedingungen durch die Entwickler versichert wurde. Um das 5. Problem zu verhindern, wird von Entwicklern beim Aufruf der Funktionen zusätzlich eine Begründung verlangt, warum die Vorbedingungen eingehalten werden.

In Abschnitt 2 wird zunächst die Terminologie besprochen, die in dieser Arbeit verwendet wurde. Abschnitt 3 beschäftigt sich mit der konkreten Implementierung von *pre* und einigen Problemen, die dabei aufgetreten sind. In Abschnitt 4 wird schlussendlich mithilfe eines qualitativen Ansatzes erforscht, wie nützlich ein Werkzeug wie *pre* ist und wie es von Rust-Entwicklern angenommen wird.

1.1 Verwandte Arbeiten

Es gibt viele Arbeiten, die sich mit vertragsbasierter Entwicklung[7] beschäftigen [4–7]. Auch für Rust gibt es eine Bibliothek, die vertragsbasierte Entwicklung ermöglicht [8]. All diese Arbeiten konzentrieren sich jedoch auf die Überprüfung der Verträge zur Laufzeit, während in dieser Arbeit der Fokus auf Überprüfungen während des Kompilierens gelegt wird.

Weiterhin wird auch an verschiedenen Methoden geforscht, um den Umgang mit unsicherem Code in Rust sicherer zu machen. So gibt es unter anderem Arbeiten

- zur automatischen Suche nach möglichen falschen Verwendungen von Zeigern in unsicherem Code [9, 10].
- zu formalen Beweisen von Teilen von Rust [11].
- zur Formalisierung des Speichermodells für Referenzen und Zeiger [12].

Auch in diesen wird versucht, eine Möglichkeit zu bieten besser mit unsicherem Code umzugehen, allerdings wird ein anderer Ansatz verwendet.

Diese Arbeit wurde unter anderem inspiriert von zwei Rust-Bibliotheken, die das Angeben von Vorbedingungen an der Definition von unsicheren Funktionen [13] und

[7] Design by contract

beim Verwenden von unsicherem Code [14] ermöglichen. Diese funktionieren jedoch völlig unabhängig voneinander und zur Laufzeit, während diese Arbeit versucht, ihre Funktionalitäten zu kombinieren und die Überprüfungen während des Kompilierens durchzuführen.

Es gibt auch für andere Sprachen Werkzeuge, die mit Annotationen arbeiten, um falsche Annahmen zu finden. Mithilfe von Splint [15] oder SAL[8] [16, 17] Annotationen an einer Funktionsdefinition können automatisch Fehler in der Implementierung oder Verwendung der Funktion gefunden werden. Diese Werkzeuge arbeiten auf einer semantischen Ebene mit den Sprachen und den Annotationen. Hier konzentriere ich mich stattdessen auf syntaktische Überprüfungen und überlasse die semantische Prüfung den Entwicklern.

2 Terminologie

In diesem Abschnitt werde ich einige Rust-spezifische Begriffe erklären. Außerdem werde ich bei mehrdeutigen Begriffen verdeutlichen, wie sie in dieser Arbeit zu verstehen sind.

2.1 Unsicherer Code

Unsicherer Code ist hier genauso definiert wie in der Rust-Referenz [18]: Es ist Code, der *potenziell* die Speichersicherheit gefährden kann. Die Verwendung des Wortes unsicher könnte vermuten lassen, dass Code gemeint ist, der dies auch tatsächlich tut. Dies ist hier *nicht* gemeint.

2.2 „stable“/„nightly“ Compiler

Es gibt verschiedene Versionen des Rust-Compilers. Der „stable“ Compiler wird alle sechs Wochen veröffentlicht. Code, der einmal erfolgreich mit einem „stable“ Compiler kompiliert, sollte auch mit zukünftigen Versionen des „stable“ Compilers kompilieren. Vom sogenannten „nightly“ Compiler wird jeden Tag eine neue Version veröffentlicht. In diesen Versionen werden neue Features veröffentlicht und Benutzern zum Testen zur Verfügung gestellt. Diese neuen Features können sich jedoch im Laufe der Zeit noch ändern oder auch wieder entfernt werden [19].

2.3 Trait

Ein Trait bietet in Rust die Möglichkeit, das Verhalten von generischen Typen zu beschreiben. Traits sind vergleichbar zu Typklassen in Haskell oder Schnittstellen in Java.

2.4 Item

Items sind Codekonstrukte, die direkt in einem Rust-Modul existieren können. Dazu gehören Funktionen, Module, Traits, Strukturen und weitere Komponenten. Eine

[8] Microsoft source code annotation language

komplette Liste ist in der Rust-Referenz zu finden [20].

2.5 ZST (Zero-Sized Type)

Ein ZST ist ein Typ, der eine feste Größe von `0` Bytes im Speicher hat. Somit kann die Verwendung eines solchen Typs oft komplett weg optimiert werden [21].

2.6 Attribute

Attribute sind in Rust Annotationen, die an verschiedenen Codekonstrukten angebracht werden können. Beispielsweise ist `#[inline(never)]` in `#[inline(never)] fn foo() {}` ein Attribut an der Funktion `foo`. Attribute können unterschiedliche Effekte haben.

2.7 Funktionen, assoziierte Funktionen und Methoden

Je nach verwendeter Programmiersprache sind die Begriffe Funktion und Methode nicht immer eindeutig. In Rust werden Funktionen mit einem `self`-Parameter[9] als Methoden bezeichnet. Diese können nur im Kontext von einem Implementierungsitem definiert werden [22]. Innerhalb eines solchen Items können auch Funktionen ohne `self`-Parameter definiert werden. Diese werden als assoziierte Funktionen bezeichnet. Alle anderen Funktionen werden nur als Funktionen bezeichnet.

3 Implementierung

```
#[pre(valid_ptr(zeiger, w))]
unsafe fn schreibe_null(zeiger: *mut u8) {
    #[assure(
        valid_ptr(dst, w),
        reason = "zeiger ist ein gueltiger Zeiger wegen der
           Vorbedingung von 'schreibe_null'"
    )]
    #[assure(
        proper_align(dst),
        reason = "'u8' hat eine Ausrichtung (alignment) von
           1 (was jeder Zeiger hat)"
    )]
    schreibe_zeiger(zeiger, 0)
}
```

Abbildung 1: Beispielverwendung von *pre*

[9] Ein spezieller Parameter mit dem Namen `self`, der nur bestimmte Typen annehmen kann und der erste Parameter sein muss.

In Abbildung 1 wird eine Beispielanwendung von *pre* gezeigt. In dem Beispiel hat die Funktion `schreibe_null` die Vorbedingung, dass `zeiger` ein zum Schreiben[10] gültiger Zeiger sein muss. In `schreibe_null` wird eine unsichere Funktion `schreibe_zeiger`[11] aufgerufen. Um anzugeben, dass die Vorbedingungen von `schreibe_zeiger` eingehalten wurden, werden `assure`-Attribute verwendet, in denen die Vorbedingungen angegeben werden. Zusätzlich muss in dem `assure`-Attribut noch ein Grund angegeben werden, warum die Bedingung gilt. `dst` ist hier der Name des Zeigerarguments der nicht gezeigten `schreibe_zeiger` Funktion.

Es gibt verschiedene Arten von Vorbedingungen in *pre*:

- `valid_ptr`, um anzugeben, dass ein Zeiger zum Schreiben, zum Lesen oder für beides gültig[12] ist.
- `proper_align`, um anzugeben, dass ein Zeiger eine für seinen Typ passende Zeigerausrichtung hat.
- Ein beliebiger boolescher Ausdruck, der wahr sein muss, damit die Vorbedingung eingehalten wird[13].
- Eine beliebige Zeichenkette, die die Vorbedingung beschreibt[14].

3.1 Grundlegender Implementierungsansatz

Es gibt verschiedene Möglichkeiten, ein Werkzeug wie *pre* zu implementieren.

Eine Möglichkeit wäre, den Compiler zu erweitern, um die Vorbedingungen zu erkennen und zu überprüfen. Dieser Ansatz erlaubt am meisten Flexibilität, da auf alle nötigen Informationen direkter Zugriff besteht. Allerdings müsste das Werkzeug sich dann allen Codeänderungen des Compilers anpassen, um auf dem neusten Stand zu sein[15], was mit einem hohen Wartungsaufwand verbunden wäre.

Eine weitere Möglichkeit wäre, ein externes Programm zu entwickeln, welches den Quelltext analysiert und die Vorbedingungen und ihre Begründungen betrachtet. Dieser Ansatz ist jedoch mit einem hohen Aufwand verbunden, da der Quelltext so weit analysiert werden muss, bis deutlich ist, wo welche Funktion aufgerufen wird. Die Mischung aus einem fehlenden globalen Funktionsnamensraum, der Möglichkeit, Funktionen als Werte in Variablen zu speichern, und der Verwendung von Typinferenz in Rust machen diese Aufgabe sehr komplex. Dieser Implementierungsaufwand wäre im Zeitrahmen einer Bachelorarbeit nicht machbar, da aufgrund des Zusammenspiels dieser Mechanismen fast die komplette Syntax und Semantik von Rust durch das Werkzeug verstanden werden müsste.

[10] Das `w` steht hier für „writable“.

[11] Diese wird hier nicht gezeigt, ist jedoch äquivalent zu `std::ptr::write` mit durch *pre* definierten Vorbedingungen.

[12] Nach der Definition in der Standardbibliothek: [23]

[13] Der Ausdruck muss nicht tatsächlich evaluiert werden. Ein Beispiel für einen solchen Ausdruck wäre `len * mem::size_of::<T>() <= isize::MAX`.

[14] Ein Beispiel hierfür wäre `"Der Inhalt von ‘v‘ ist gueltiges UTF-8"`.

[15] Alternativ müsste das Werkzeug direkt in den offiziellen Compiler übernommen werden, was jedoch die Ansprüche einer Bachelorarbeit übersteigen würde.

Für die Implementierung in dieser Arbeit wurden schlussendlich prozedurale Attribut-Makros [24] verwendet. Attribute können in Rust an verschiedene Code-Konstrukte geschrieben werden, um diese zu beeinflussen. Durch die Verwendung von Makros zur Implementierung des Konzeptes ist es möglich, *pre* direkt als normale Rust-Bibliothek [3] zu verwenden.

Prozedurale Attribut-Makros erlauben es, während des Kompilierens den Quelltext von annotierten Items[16] zu inspizieren und zu bearbeiten. Hierfür wird eine normale Funktion in Rust mit folgender Signatur geschrieben:

```
fn attribut_name(item_tokens: TokenStream, attribut_tokens:
    TokenStream) -> TokenStream;
```

Diese erhält als Argumente die Tokens[17] des Items und der Attributsparameter und kann beliebige Tokens ausgeben. Diese werden dann vom Compiler statt des ursprünglichen Items verwendet. Somit kann der Code einer annotierten Funktion beliebig verändert werden, jedoch kann dabei nur auf Informationen, die in der Funktionsdefinition selbst und im Attribut vorhanden sind, zugegriffen werden. Aus diesem Grund ist es im allgemeinen Fall unmöglich, von einem Attribut an einem Funktionsaufruf auf die an der Definition vorgehaltenen Vorbedingungen zurückzugreifen. Stattdessen werden die Vorbedingungen als Typ in einem zusätzlichen Funktionsargument codiert. Dadurch überprüft der Compiler, ob die Vorbedingungen gleich sind, während er die Typen der Funktionsargumente überprüft. Die Herausforderung bei diesem Ansatz ist, eine passende Codierung der Vorbedingungen zu finden.

3.2 Codierung der Vorbedingungen in einem Typ

Eine Codierung der Vorbedingungen als Typ sollte folgende Eigenschaften haben:

1. Es sollte möglich sein, beliebig viele Vorbedingungen in einem einzigen Typ zu codieren. Dies sorgt für eine bessere Fehlermeldung bei einem Funktionsaufruf mit fehlenden Vorbedingungen.

2. Der Typ muss an der Definition der Funktion mit Vorbedingungen benennbar sein.

3. Es muss beim Aufruf der Funktion mit Vorbedingungen möglich sein, Werte des Typs zu erstellen.

4. Da unter anderem beliebige Zeichenketten als Vorbedingung möglich sind, muss die Codierung in der Lage sein, diese zu codieren.

5. Es muss möglich sein, unterschiedliche Vorbedingungstypen zu codieren. Wenn Eigenschaft 4 gegeben ist, kann dies jedoch über eine Codierung der Vorbedingung als Zeichenkette mit einem Präfix, der den Typ beschreibt, gelöst werden.

[16] siehe Definition 2.4 (Seite 12)
[17] Der Begriff bezeichnet hier die Ausgabe des Lexers.

6. Der Typ muss ein ZST[18] sein, damit jedes Vorkommen von ihm ohne Weiteres wegoptimiert werden kann.

7. Da bei einer falsch verwendeten Vorbedingung der Typ in der Fehlermeldung erscheint, ist es von Vorteil, wenn die ursprüngliche Vorbedingung leicht aus dem Typ erkennbar ist.

8. Es sollte ein Typfehler entstehen, wenn sich die Mengen der Vorbedingungen an der Funktionsdefinition und beim Aufruf unterscheiden.

In dieser Arbeit wurden zwei verschiedene Codierungen implementiert. Die eine Codierung mit Hilfe von „const generics“ hat viele Vorteile, ist jedoch nur für Nutzer des „nightly“ Compilers verfügbar. Die andere Codierung als Strukturenfelder hat einige Einschränkungen, funktioniert jedoch auch mit dem „stable“ Compiler. Bei der Kompilierung wird die Compilerversion automatisch erkannt und die bessere Version verwendet.

3.2.1 Codierung mit „const generics“

Ein momentan nur im „nightly“ Compiler unterstütztes Feature sind die sogenannten „const generics“, mithilfe derer Typen generisch über Konstanten[19] sein können. Mit „const generics“ kann in einer konkreten Instanziierung eines solchen generischen Typs die enthaltene Zeichenkette exakt codiert werden. Da „const generics“ schon länger im Compiler implementiert sind, teilweise schon von der Standardbibliothek verwendet werden und vermutlich bald zum Teil in den stabilen Compiler übernommen werden [25], ist es unwahrscheinlich, dass es noch größere Änderungen an ihnen geben wird. Deswegen und wegen der vielen Vorteile, die sie gegenüber der Implementierung mit Strukturen bieten, habe ich mich entschlossen, diese Codierung zu implementieren, obwohl „const generics“ noch nicht im „stable“ Compiler verfügbar sind.

Es ist zwar durch Typinferenz möglich, Werte eines solchen Typs zu erstellen, ohne die Zeichenkette explizit anzugeben[20], allerdings wird in dem von `assure`-Attributen generierten Code die Zeichenkette immer explizit angegeben. Dadurch entstehen Typfehler, wenn die Zeichenketten und damit auch die Vorbedingungen nicht übereinstimmen.

Da dabei alle Daten im Typ gespeichert sind, ist es möglich, diese generischen Typen als ZST zu definieren[21].

Im „nightly“ Compiler können alle Vorbedingungen als Tupel von Typen mit einem solchen generischen Parameter codiert werden. In Abbildung 2 ist ein Beispiel eines solchen Tupels zu sehen.

Ein Problem bei der Codierung als Tupel ist, dass die Reihenfolge der Vorbedingungen eine Rolle spielt. Dies lässt sich jedoch lösen, indem eine beliebige aber feste

[18] siehe Definition 2.5 (Seite 12)
[19] und damit auch Zeichenketten
[20] Für das Beispiel in Abbildung 2 wäre das mit dem Ausdruck `(:: pre :: BooleanCondition, ::pre::CustomCondition)` möglich.
[21] In Rust existieren zur Laufzeit keine Typinformationen mehr.

Ordnung für die Vorbedingungen gewählt wird und diese Ordnung für die Reihenfolge der Vorbedingungen im Tupel verwendet wird. Ein Nachteil dieses Ansatzes ist, dass die Fehlermeldungen bei Tippfehlern schwer verständlich sein können, falls der Tippfehler die Position einer Vorbedingung in dem Tupel verändert.

```
(::pre::BooleanCondition<"wert > 42.0">, ::pre::
    CustomCondition<"Bedingungstext">,)
```

Abbildung 2: Beispiel eines mit „const generics“ für die Vorbedingungen `wert > 42.0` und `"Bedingungstext"` generierten Typs. Ein vollständigeres Beispiel zum generierten Code ist in Anhang A in den Abbildungen 8 und 9 zu finden.

3.2.2 Codierung als Strukturenfelder

Da Zeichenketten in „const generics“ höchstwahrscheinlich nicht allzu bald in „stable“ Rust verfügbar sein werden[22] und der „stable“ Compiler am häufigsten genutzt wird [19], ist eine Alternative für Nutzer des „stable“ Compilers notwendig.

Eine weitere Möglichkeit mehrere Zeichenketten in einem Typ zu codieren sind Bezeichner[23]. In Bezeichnern können alphanumerische Zeichen und Unterstriche verwendet werden [26][24]. Alle anderen Zeichen müssen eindeutig durch diese Zeichen ausgedrückt werden. Dies kann beispielsweise geschehen, indem für andere Zeichen der Unicode-Codepunkt des Zeichens als Zahl codiert eingefügt wird. Außerdem kann der Vorbedingungstyp als Präfix codiert werden. So kann beispielsweise die boolesche Vorbedingung `some_val > 42.0` als `_boolean_some__val_20_3e_2042_2e0` codiert werden. Diese Codierung ist schwerer verständlich für sonderzeichen- und zahlenlastige Vorbedingungen, ist aber zumindest für textbasierte Vorbedingungen einigermaßen lesbar.

Strukturen ermöglichen, beliebig viele Felder mit jeweils eigenen Bezeichnern in einem Typ zu vereinen. Somit können alle Vorbedingungen in einer einzigen Struktur als jeweils ein Feld codiert werden. Für alle Felder wird ein ZST als Typ gewählt und somit ist auch die gesamte Struktur ein ZST. Ein Beispiel für eine solche Struktur ist in Abbildung 3 zu finden.

```
struct foo {
    _boolean_wert_20_3e_2042_2e0: (),
    _custom_Bedingungstext: (),
}
```

Abbildung 3: Beispiel eines mithilfe einer Struktur für die Vorbedingungen `wert > 42.0` und `"Bedingungstext"` generierten Typs. Ein vollständigeres Beispiel zum generierten Code ist in Anhang A in den Abbildungen 8 und 10 zu finden.

[22] Für Ganzzahlen werden sie wahrscheinlich in naher Zukunft verfügbar sein [25].

[23] Mit Bezeichnern sind hier Namen von Codekonstrukten in Rust gemeint, wie in der Referenz beschrieben [26].

[24] Es wird jedoch daran gearbeitet, weitere Zeichen zuzulassen [27].

Um einen Wert dieser Struktur erstellen zu können, müssen alle Felder mit ihrem vollständigen Bezeichner genannt werden. Wenn nicht jeder Bezeichner exakt einmal genannt wurde, erzeugt der Compiler eine Fehlermeldung.

Ein Problem, das bei der Verwendung von Strukturen auftritt, ist, dass diese bei ihrer Verwendung benennbar sein müssen. Dieses Problem lässt sich lösen, indem der Struktur der gleiche Name wie der Funktion gegeben wird [28][25], um so die Struktur automatisch mit der Funktion gemeinsam zu importieren. Da Strukturnamen sich im Typ-Namensraum und Funktionsnamen sich im Wert-Namensraum befinden, können diese parallel über die gleiche Importierungsanweisung importiert werden. Diese Lösung funktioniert allerdings nicht, wenn sich aus einem anderen Grund im Typ-Namensraum bereits der Name der Funktion befindet. Dies ist in der Praxis allerdings unwahrscheinlich.

Ein zusätzliches Problem tritt auf, wenn Vorbedingungen für Methoden und assoziierte Funktionen angegeben werden sollen. Da diese in einem Implementierungsitem definiert werden und in einem solchen keine Strukturen definiert werden können, gibt es für das Makro keinen Ort, an dem die passende Struktur generiert werden könnte. Somit ist das Angeben von Vorbedingungen für Methoden und assoziierte Funktionen mit diesem Ansatz nicht möglich.

Dieser Ansatz funktioniert ebenfalls nicht, wenn die aufgerufene Funktion über eine Variable umbenannt wird oder sogar nicht direkt lokal benannt wird[26]. In diesen Fällen wäre der Name der Struktur entweder falsch oder unmöglich herauszufinden.

3.3 Definition von Vorbedingungen für Fremdcode

pre kann für interne Funktionsaufrufe innerhalb eines Projektes verwendet werden. Der Hauptanwendungsfall ist jedoch die Überprüfung der Vorbedingungen von Fremdcode. Hier ist die Wahrscheinlichkeit, dass Fehler entstehen deutlich größer, da Entwickler in der Regel mit Fremdcode nicht so vertraut sind wie mit ihrem eigenen Code. Eine besonders wichtige Komponente ist hier die Standardbibliothek, in der viele häufig genutzte unsichere Funktionen definiert sind. *pre* bietet für einige unsichere Standardbibliotheksfunktionen äquivalente Funktionen, die zusätzlich Vorbedingungen haben.

Um Vorbedingungen für die Standardbibliothek und anderen Fremdcode angeben zu können, werden neue Funktionen definiert, die die gleiche Signatur haben. Die Vorbedingungen werden an diesen Funktionen angegeben. Innerhalb dieser Funktionen wird nun die ursprüngliche Funktion aufgerufen. Durch Verwendung des `#[inline(always)]` Attributs wird dem Compiler empfohlen, den zusätzlichen Funktionsaufruf weg zu optimieren, sodass zur Laufzeit kein Mehraufwand entsteht. Wenn die Vorbedingungen einer Funktion überprüft werden sollen, wird statt der ursprünglichen Funktion die neue Funktion aufgerufen. Um das Erstellen solcher Funktionen zu erleichtern, wurde ein Helfermakro zu diesem Zweck erstellt. In Abbildung 4 ist ein Beispiel für das Helfermakro und den Ausgabecode zu sehen.

[25] In der Quelle wird ein anderes Problem beschrieben, jedoch ist die Lösung dieses Problems auch hier anwendbar.

[26] Dies kann beispielsweise der Fall sein, wenn eine Funktion aus einer Liste von Funktionen aufgerufen wird.

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

// Resultierender Code (vereinfacht)
mod pre_std {
    pub(crate) mod ptr {
        #[pre(valid_ptr(dst, w))]
        #[inline(always)]
        pub(crate) unsafe fn write_unaligned<T>(dst: *mut T,
            src: T) {
            std::ptr::write_unaligned(dst, src)
        }
    }
}
```

Abbildung 4: Beispiel für das `extern_crate` Helfermakro. Ein vollständigeres Beispiel zum generierten Code ist in Anhang A in den Abbildungen 12 und 13 zu finden.

3.3.1 Methoden in Fremdcode

Eine Kopie mit Vorbedingungen zu erstellen ist für Funktionen möglich, jedoch nicht für Methoden. Methoden können in Rust nicht für fremde Typen definiert werden. Alternativ wäre es möglich eine Funktion, die den `self` -Parameter als ersten Parameter bekommt und intern die Methode aufruft, zu erstellen. Es ist jedoch nicht immer möglich, diese korrekt aufzurufen, da der genaue Typ des `self` -Parameters bei einem Methodenaufruf nicht erkennbar ist. Ein Methodenaufruf `x.foo()` für eine Methode `fn foo(&self) {}` definiert auf einem Typ `X`, wird vom Compiler in einen Funktionsaufruf `X::foo(&x)` umgewandelt. Der gleiche Methodenaufruf wird jedoch zu `X::foo(x)` umgewandelt, wenn die Signatur der Methode zu `fn foo(self) {}` geändert wird. Man beachte, dass es sich einmal um eine Referenz handelt und einmal nicht. Dies sind in Rust völlig unterschiedliche Typen. Es ist also im allgemeinen Fall nicht möglich, den genauen Typ des `self` -Parameters zu kennen, ohne auch die Signatur der Methode zu kennen. Der Algorithmus, der bestimmt wie der erste Parameter übergeben wird, wird nur bei Methodenaufrufen durchlaufen [29], somit kann eine Funktion eine Methode hier nicht ersetzen. Es gibt verschiedene andere Möglichkeiten, Vorbedingungen für Methoden zu definieren, diese haben jedoch alle größere Nachteile.

Eine Möglichkeit wäre, eine neue Struktur zu erstellen, die nur ein Feld mit dem

Typ, auf dem die Methode ursprünglich definiert war, enthält[27]. Diese müsste dann alle Methoden des ursprünglichen Typs kopieren, um dann lokal fast genauso wie der ursprüngliche Typ zu funktionieren. Allerdings müsste die Struktur ebenso alle Traits[28] des ursprünglichen Typs implementieren. Außerdem wäre es nicht möglich, diese Struktur an Funktionen zu übergeben, die den ursprünglichen Typ als Parameter haben. All diese Einschränkungen sorgen dafür, dass dieser Ansatz nur in manchen Situationen tatsächlich verwendet werden könnte, weshalb er in dieser Arbeit nicht implementiert wurde.

Eine andere Möglichkeit wäre, eine parameterlose leere Funktion zu definieren, die die Vorbedingungen der Methode bekommt. Ein Methodenaufruf bleibt bei diesem Ansatz unverändert, wird jedoch um einen Aufruf der leeren Funktion mit den entsprechenden Vorbedingungen ergänzt[29]. Dieser Ansatz hat den Nachteil, dass bei jedem Methodenaufruf der Pfad zu der parameterlosen Funktion bekannt sein und damit manuell angegeben werden muss. Ein weiterer Nachteil ist, dass die ursprüngliche Methode völlig unverändert ist und somit fehlende Vorbedingungen bei einem Aufruf nicht bemerkt werden können. Der Aufruf muss erst manuell mit der Angabe des Pfades zur parameterlosen Funktion in einen überprüften Aufruf umgewandelt werden[30]. Vorteilhaft bei dieser Herangehensweise ist jedoch, dass sie auch mit dem „stable“ Compiler funktioniert, da die Vorbedingungen an einer Funktionsdefinition angegeben werden und nicht an einer Methodendefinition oder der Definition einer assoziierten Funktion. Dies erlaubt es Vorbedingungen von unsicheren Methodenaufrufen von Methoden der Standardbibliothek auch mit dem „stable“ Compiler zu überprüfen. Deshalb ist dieser Ansatz auch die Lösung, die für diese Arbeit gewählt wurde. Ein vollständiges Beispiel des hierfür generierten Codes ist in Anhang A in den Abbildungen 12 und 13 zu finden.

3.4 Attribute an Ausdrücken

In Rust ist es momentan noch nicht möglich, Ausdrücke mit Attributen zu versehen [31]. Somit können die `assure`-Attribute, die *pre* anbietet, nicht direkt verwendet werden. Eine Möglichkeit, dies zu umgehen, wurde von David Tolnay beschrieben [32]. Er schlägt vor, die Funktion, die den Ausdruck enthält, mit einem Attribut zu versehen. Dieses äußere Attribut kann dann die gesamte Funktion nach Ausdrücken mit dem entsprechenden inneren Attribut durchsuchen, das innere Attribut entfernen und den Ausdruck entsprechend anpassen. Hierfür bietet die syn-Bibliothek eine Möglichkeit, alle Ausdrücke in einer Funktion nacheinander zu inspizieren und gegebenenfalls zu verändern [33]. In diesem Schritt werden auch mehrere `pre`- und `assure`-Attribute zu einer Liste von Vorbedingungen zusammengefasst.

[27] Dies wird in Rust als „newtype“-Pattern bezeichnet [30].

[28] siehe Definition 2.3 (Seite 11)

[29] Aus `x.foo()` wird also zu `#[assure(/* ... */)] leere_funktion_mit_vorbedingungen(); x.foo()` äquivalenter Code. Im Anhang A in den Abbildungen 18, 19 und 20 ist der in einer solchen Situation tatsächlich generierte Code zu sehen.

[30] Dies würde im Code mithilfe eines `forward`-Attributs funktionieren: `#[forward(impl pfad::zu::funktion)] #[assure(/* ... */)] x.foo()`

3.5 Status der Implementierung

Die Implementierung aller hier beschriebenen Bestandteile wurde abgeschlossen. Die dabei wichtigsten Bibliotheken waren syn [34], quote [35] und proc-macro-error [36]. Für alle implementierten Teile gibt es außerdem automatische Tests der Compilerausgaben, die mithilfe der trybuild Bibliothek [37] implementiert wurden.

Andererseits gäbe es noch viele kleine Anpassungen, die die Implementierung besser machen würden, für die ich im Rahmen der Arbeit keine Zeit mehr gefunden habe. Außerdem sollten die Vorbedingungen für die unsicheren Standardbibliotheksfunktionen vervollständigt werden. Davon abgesehen ist die Implementierung meiner Meinung nach in einem guten Zustand und durchaus schon in der Praxis nutzbar.

4 Evaluation

In diesem Abschnitt soll evaluiert werden, wie nützlich die im Abschnitt „Implementierung“ entstandene Bibliothek ist. Es sollen unter anderem folgende Fragen beantwortet werden:

1. Wann ist eine solche Bibliothek hilfreich und wann nicht?
2. Besteht Interesse an einer solchen Bibliothek?
3. Rechtfertigt die gewonnene zusätzliche Sicherheit den Mehraufwand?
4. Was ist noch verbesserungswürdig?

Um diese Fragen zu beantworten, habe ich online Beiträge über die Bibliothek veröffentlicht und um Feedback gebeten. Außerdem habe ich nach quelloffenen Rust-Projekten gesucht, die unsicheren Code verwenden und potenziell von einer Lösung wie *pre* profitieren könnten. Diesen habe ich angeboten, *pre* in ihr Projekt zu integrieren[31].

4.1 Projektintegration

In diesem Abschnitt werden die notwendigen Überlegungen zur Integration von *pre* in ein Projekt getroffen. Daraufhin wird diese am Beispiel der Bibliothek „secstr“ beschrieben.

4.1.1 Verwendungsmöglichkeiten von *pre* in Projekten

Die Integration von *pre* in ein Projekt kann in zwei Stufen unterteilt werden: interne Verwendung und externe Verwendung.

Bei der internen Verwendung werden `assure`-Attribute an Aufrufe unsicherer Funktionen innerhalb der Bibliothek geschrieben und begründet, warum die Aufrufe sicher sind. Hierbei ist der Prozess des Verfassens häufig wichtiger als das Resultat, weil beim Verfassen eine Prüfung des Codes auf korrekte Verwendung durchgeführt

[31] Bei größeren Projekten habe ich es nur für einen Teil des Projektes angeboten, den die Autoren benennen konnten.

wird. *pre* ist dabei nur ein Werkzeug, das auf Stellen hinweist, an denen noch nicht geprüft wurde. Das Resultat ist dann eine Dokumentation dieses Prüfungsprozesses, die in den Quelltext mit aufgenommen werden kann, falls dies erwünscht ist.

Die externe Verwendung erfolgt, wenn ein Projekt Vorbedingungen von *pre* in die öffentlich erreichbaren unsicheren Funktionen mit aufnimmt. Damit werden die Vorbedingungen Teil der API des Projektes und Nutzer müssen sich mit den Vorbedingungen auseinandersetzen.

In der Projektsuche habe ich mich hauptsächlich auf die interne Verwendung in Projekten fokussiert, da dies für die Projektverwalter ein deutlich kleinerer Schritt ist, als die API ihres Projektes zu verändern.

4.1.2 Projektsuche

Bei der Projektsuche gab es einige Kriterien, auf die ich geachtet habe, bevor ich mich entschieden habe, bei dem Projekt anzufragen:

- Das Projekt sollte mindestens fünf bis zehn unsichere Funktionsaufrufe enthalten, sonst hätte es sich nicht sonderlich gut geeignet, um die Nützlichkeit von *pre* zu evaluieren.
- Das Projekt sollte nicht zu komplex sein, da ich im Rahmen der Bachelorarbeit nur begrenzt Zeit hatte, mich in das Projekt einzuarbeiten. Aus dem gleichen Grund sollte das Projekt nicht zu viel Domänenwissen bei der Einarbeitung erfordern.
- Die Sicherheit des unsicheren Codes im Projekt sollte sich nicht auf komplexe Invarianten stützen, da *pre* für diesen Fall nicht gut geeignet ist.
- Das Projekt sollte mindestens zehn Downloads pro Tag haben. Ein Projekt, das diese Voraussetzung nicht erfüllt, hätte sich schlechter geeignet, um zu beurteilen, wie *pre* angenommen wird.

Insgesamt habe ich auf meiner Suche nach passenden Projekten fünf Projekte kontaktiert [38–42]. Es gab jedoch nur bei einem dieser Projekte eine Rückmeldung.

Auf der Suche nach Projekten fiel mir schon das erste Problem bei *pre* auf. Viele Entwickler wollen mehr Abhängigkeiten eher vermeiden. Bei einem Projekt bestand sogar eine Regel, keine Abhängigkeiten unter Version 1.0.0 aufzunehmen [43].

Aus diesem Grund und damit Nutzer von Bibliotheken, die *pre* nutzen nicht längere Kompilationszeiten haben müssen, habe ich *pre* daraufhin noch um eine Möglichkeit erweitert, optional verwendet zu werden. Mit sogenannten „Konfigurationsprädikaten“ [44] ist es möglich die Ausführung der Attribut-Makros von *pre* zu erlauben oder zu verhindern. Dies wird zwar normalerweise vom Compiler übernommen, allerdings muss *pre* mit diesen Prädikaten gesondert umgehen, da *pre* seine Annotationen teilweise selber analysiert.

Eine solche Möglichkeit zu bieten, erlaubt es Nutzern, das Überprüfen der Vorbedingungen zu umgehen, und kann damit aktiv dem Ziel von *pre* entgegenwirken. Es ist Nutzern jedoch auch ohne eine solche Option möglich, die Vorbedingungen nicht

zu überprüfen, indem sie einen beliebigen Grund angeben[32], ohne weiter darüber nachzudenken. *pre* zielt darauf ab, in der Praxis verwendbar zu sein und eine solche Option erlaubt die Nutzung von *pre* in mehr Fällen. So können Entwickler beispielsweise lokal Rechenzeit sparen, indem sie *pre* deaktivieren, während *pre* auf einem Testserver aktiviert ist und somit sicherstellt, dass alle Vorbedingungen letztendlich eingehalten werden.

4.1.3 „secstr“

„secstr“ ist eine Rust-Bibliothek, die einen Datentyp zum Aufbewahren sensibler Daten implementiert [45]. Der Autor von „secstr“ hat zugestimmt, dass er an einer Integration von *pre* in „secstr“ interessiert ist. Zu diesem Zeitpunkt hatte „secstr“ neun Verwendungen von unsicheren Standardbibliotheksfunktionen. Zwei davon konnten trivial durch sicheren Code ersetzt werden.

Außerdem gab es noch einige Funktionsaufrufe zu in C verfassten Funktionen[33]. Da ich mich nicht genug mit den verwendeten C-Funktionen auskenne, habe ich diesen unsicheren Code jedoch nicht weiter beachtet. Generell ist die Verwendung von *pre* für nicht-Rust Code erschwert, da kaum andere Sprachen ein Konzept von unsicheren Funktionen haben und damit in der Regel einzuhaltende Vorbedingungen nicht ausführlich kommentiert werden.

Um die Vorbedingungen der Standardbibliotheksfunktionen angeben zu können, musste ich für diese erst Vorbedingungen in *pre* hinzufügen. Dabei sind mir einige Fehler in der Standardbibliotheksdokumentation aufgefallen, für die ich Pull Requests eröffnet habe [46–49].

Beim Überprüfen der Vorbedingungen innerhalb von „secstr“ mithilfe von *pre*, konnte ich an einer Stelle zuvor unbekanntes undefiniertes Verhalten finden. Dieses trat in der `mem::cmp` Funktion auf, welche beliebige Listen eines generischen Typs in konstanter Zeit auf Gleichheit überprüfen soll, indem diese als Bytearrays verglichen werden. Hierfür wurden Zeigerzugriffe verwendet, um die einzelnen Bytes der Listen zu lesen. Dies kann auch durch die unsichere Funktion `std::ptr::read` erreicht werden, wobei jedoch die Vorbedingungen für den Zeigerzugriff überprüft werden können. Eine der Vorbedingungen von `std::ptr::read` ist, dass der Zeiger, der als Argument übergeben wird, auf einen gültigen und initialisierten Wert zeigen muss. In diesem Fall konnte allerdings nicht garantiert werden, dass jedes Byte in der Liste initialisiert ist, da die Listen generisch sind und uninitialisierte Füllbytes[34] enthalten können. Dies konnte ich erkennen, da das Einhalten der Vorbedingung nicht in allen Fällen begründet werden konnte. Dieser Fehler kann unter anderem dazu führen, dass zwei semantisch gleiche Objekte als ungleich verglichen werden, da sie unterschiedliche Werte in ihren Füllbytes haben. Nachdem ich zwei Lösungsvorschläge unterbreitet habe [50], habe ich eine Lösung implementiert, die Vergleiche dieser Art nur für Typen ohne Füllbytes erlaubt.

Im Laufe meiner Arbeit an „secstr“, habe ich auch einen schon bekannten Fehler

[32] `"42"` bietet sich beispielsweise an.

[33] Funktionsaufrufe von nicht in Rust geschriebenen Funktionen sind in Rust immer unsicher, da die anderen Sprachen die Invarianten von Rust beliebig verletzen können.

[34] Padding bytes

behoben. Der Fehler trat in einer Funktion auf, die den Speicher eines generischen Typs mit `0`-Bytes[35] überschreiben sollte. Das Problem war, dass der generische Typ danach immer noch als initialisiert galt, es jedoch konkrete Typen gibt, für die ein mit `0`-Bytes gefüllter Speicher kein gültiger Wert ist. Konkret entstand der Fehler durch eine Typumwandlung von einem generischen Zeiger `⋆mut T` in einen konkreten Zeiger `⋆mut u8`. Dies ist in Rust eine Operation, die auch außerhalb von unsicherem Code durchgeführt werden kann, jedoch nur in unsicherem Code zu Problemen führen kann [51]. Aus diesem Grund hätte dieser Fehler nicht mithilfe von *pre* gefunden werden können.

Das Ergebnis der Integration von *pre* in „secstr“ ist in Form eines Pull Requests in den Quelltext von „secstr“ aufgenommen worden [52].

4.2 Wann ist eine solche Bibliothek hilfreich und wann nicht?

Eine Bibliothek wie *pre* ist nicht sonderlich hilfreich, wenn sich die Sicherheit einer Implementierung auf die Einhaltung von komplexen Invarianten stützt. In diesem Fall sind die Gründe für das Einhalten der Vorbedingungen nicht lokal und somit ist eine lokale Begründung nicht ausreichend. Stattdessen sind Zusicherungen[36] in Kombination mit Stresstests in solchen Fällen ein besseres Werkzeug.

pre kann deutlich hilfreicher sein, wenn es sich um Vorbedingungen handelt, die mit lokal vorhandenen Informationen begründet werden können. In diesem Fall ermöglicht es Entwicklern, schnell unsichere Funktionsaufrufe zu erkennen und zu überprüfen, ob diese wirklich sicher sind. Außerdem ermöglicht *pre* das Angeben und manuelle Überprüfen von Vorbedingungen, die nicht automatisch zur Laufzeit überprüft werden, wie beispielsweise die Gültigkeit von Zeigern.

4.3 Besteht Interesse an einer solchen Bibliothek?

Der Autor von „secstr“ hat nach der Integration von *pre* folgendes Feedback zu *pre* abgegeben [52][37]:

> „hm.. well I haven't actually used pre, just briefly looked at the code you changed here.
>
> I like the idea of making developers manually write out why preconditions are satisfied, I don't think anything other than the compiler absolutely requiring this would make me think carefully about everything :D
>
> It would be nice if the same preconditions could be also used for auto-generating runtime checks (like in D [53]), for deductive verification (like Frama-C WP [54] / Ada SPARK), and so on. Really I guess Rust needs to eventually get native contract syntax like in D that would be usable by all the tools including pre (and it should be informed by pre's use case).“

[35] Gemeint sind hier Bytes, die den Wert `0x00` haben.

[36] Mithilfe des `assert` oder `debug_assert` Makros

[37] Die Referenzen sind vom Autor

Außerdem gab es auf Beiträge, in denen *pre* erläutert wurde, die Rückmeldungen „This is a really cool idea. Nice work!“ [55] und „Looks like a nice idea.“ [56]. Gegenmeinungen hierzu gab es als Rückmeldungen zu diesen Beiträgen nicht.

Es besteht also zumindest unter denjenigen, die Feedback gegeben haben, Interesse an einer solchen Bibliothek und eventuell sogar an einer Integration in den Compiler.

4.4 Rechtfertigt die gewonnene zusätzliche Sicherheit den Mehraufwand?

Um diese Frage quantitativ beantworten zu können, müsste es mehr Nutzer von *pre* geben. Daraufhin müsste eine Methode gefunden werden, mithilfe derer geprüft werden kann, ob in von *pre* annotiertem Code tatsächlich weniger Fehler auftreten. Außerdem müsste ein Maß für das Ausmaß der Fehler festgelegt werden. Hier wird jedoch stattdessen eine qualitative Antwort gegeben.

Ob es sich lohnt, *pre* zu verwenden, hängt stark vom Projekt ab. Es gibt Projekte, die ähnliche Anmerkungen wie *pre*-Annotationen in Kommentaren machen [57], für die es vermutlich wenig zusätzlichen Aufwand gegeben hätte, wenn sie *pre* von Anfang an verwendet hätten. Die korrekte Verwendung von *pre* kann bestimmte Arten von menschlichem Versagen verhindern. Der Nutzen hängt also von der Wahrscheinlichkeit ab, mit der dieses Versagen auftritt. Meine Vermutung ist, dass der Mehraufwand mit der Versagenswahrscheinlichkeit positiv korreliert. Entwickler, die sich viele Gedanken zur Sicherheit in ihrem Code machen, haben vermutlich eine geringere Wahrscheinlichkeit Fehler zu machen. Sie sind die nötigen Begründungen in ihren Gedanken schon durchgegangen. Für diese Entwickler wäre der Mehraufwand vermutlich eher gering, jedoch ebenso der Nutzen. Für Entwickler, die sich keine Gedanken zur Sicherheit ihres Codes machen, wäre der Mehraufwand vermutlich deutlich größer, aber der Nutzen dementsprechend auch.

4.5 Was ist noch verbesserungswürdig?

Ein Kritikpunkt war, dass Zeichenketten als Vorbedingungen keine gute Lösung sind [56]. Hierfür gibt es in *pre* bereits verschiedene Vorbedingungstypen, die auch über Zeichenketten hinausgehen. Es könnte noch mehr Varianten davon geben und diese sollten in Beispielen mehr verwendet werden, um auf ihre Existenz aufmerksam zu machen.

Ein weiterer Kritikpunkt war die Auswahl des Begriffs „assure“ für die Annotationen, mit denen das Einhalten der Vorbedingungen begründet wird[38]. Hier könnten die verschiedenen Begriffsoptionen erneut evaluiert werden, um potenziell einen besseren Begriff zu finden. Um dem Problem entgegenzuwirken, wurde jedoch ein Abschnitt in der Dokumentation von *pre* der Wahl des Begriffs „assure“ gewidmet.

Bei der Suche nach Projekten, in die ich *pre* integrieren könnte, ist mir aufgefallen, dass *pre* nicht gut für unsicheren Code geeignet ist, der sich auf Invarianten für seine Sicherheit verlässt. Vielleicht wäre es lohnenswert, Überlegungen anzustellen, wie

[38] Der Kommentator hat seinen Kommentar diesbezüglich gelöscht, weshalb hier keine Quelle angegeben ist.

Invarianten und Nachbedingungen in das Konzept von *pre* passen könnten. Eventuell hätte *pre* sogar auf einer bestehenden Bibliothek für vertragsbasierte Entwicklung, wie „contracts“ [8], aufbauen können.

Eine der größten Einschränkungen, die *pre* momentan hat, ist die Qualität der Fehlermeldungen. *pre* hat nur eine sehr begrenzte Kontrolle über die Ausgaben des Compilers in einem Fehlerfall. In Abbildung 6 ist die Fehlermeldung gezeigt, die beim Kompilieren des Codes aus Abbildung 5 ausgegeben wird. Diese Fehlermeldung spiegelt den Idealfall wieder, da die aufgerufene Funktion und der Aufruf sich in der gleichen Datei befinden. In Abbildung 7 ist die Fehlermeldung für die gleiche `main` Funktion gezeigt, wobei sich die `run` Funktion jedoch in einer anderen Datei befindet. In diesem Fall kann die Fehlermeldung sehr verwirrend sein, da gezeigt wird, dass die Funktion anscheinend keine Parameter hat.

```
use pre::pre;

fn init() {
    /* Initialisierung durchfuehren */
}

#[pre("'init' muss aufgerufen werden, before 'run' das erste
    mal aufgerufen wird")]
fn run() {
    /* Dinge, die Initialisierung voraussetzen machen */
}

#[pre]
fn main() {
    run();
}
```

Abbildung 5: Eine Verwendung von *pre*, die eine Fehlermeldung erzeugt.

```
error[E0061]: this function takes 1 argument but 0 arguments were supplied
  --> src/main.rs:14:5
   |
7  |   #[pre("`init` muss aufgerufen werden, before `run` das erste mal aufgerufen wird")]
   |  _______-
8  | | fn run() {
9  | |     /* Dinge, die Initialisierung voraussetzen machen */
10 | | }
   | |_- defined here
...
14 |       run();
   |       ^^^-- supplied 0 arguments
   |       |
   |       expected 1 argument
```

Abbildung 6: Fehlermeldung beim Kompilieren des Codes aus Abbildung 5.

```
error[E0061]: this function takes 1 argument but 0 arguments were supplied
  --> src/main.rs:14:5
   |
14 |     run();
   |     ^^^-- supplied 0 arguments
   |     |
   |     expected 1 argument
   |
  ::: src/def_run.rs:4:1
   |
4  | pub fn run() {
   | ------------ defined here
```

Abbildung 7: Fehlermeldung bei fehlenden Vorbedingungen, wenn sich die aufgerufene Funktion in einer anderen Datei befindet.

5 Fazit

In dieser Arbeit ist das Werkzeug *pre* in Form einer Rust-Bibliothek entstanden. Es wurde festgestellt, dass es mithilfe von *pre* möglich ist, Fehler in Software zu finden. Außerdem hat sich herausgestellt, dass es Entwickler gibt, die an einem Projekt wie *pre* interessiert sind.

pre hat aber auch noch ungelöste Probleme, die den effektiven Nutzen als allgemeingültiges Werkzeug stark einschränken. Insbesondere die Fehlermeldungen von *pre* bei falschen Vorbedingungen müssten stark verbessert werden. Außerdem sollten Möglichkeiten gefunden werden, um mehr Anwendungsfälle einheitlich behandeln zu können. Dabei sollten unter anderem mehr vorgegebene Vorbedingungstypen erstellt werden, um Entwicklern eine einheitliche Sprache zum Modellieren der Vorbedingungen zur Verfügung zu stellen. Des Weiteren sollte die Funktionalität von *pre* über unsichere Funktionen hinaus auf anderen unsicheren Code erweitert werden. Eine andere mögliche Erweiterung wäre ein strikterer Modus von *pre*, in dem unsichere Funktionen ohne Vorbedingungen nicht mehr definiert und aufgerufen werden können.

pre oder ein ähnliches Werkzeug hat das Potenzial, Entwicklern eine hilfreiche Unterstützung beim Umgang mit unsicherem Code zu sein. Dieses Potenzial ist jedoch noch nicht voll ausgeschöpft.

Literaturverzeichnis

[1] "The Rustonomicon: Meet safe and unsafe." https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html. Aufruf am 01.08.2020.

[2] "The Rustonomicon: How safe and unsafe interact." https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html. Aufruf am 01.08.2020.

[3] N. Schwarzlose, "'pre' Rust Bibliothek." https://crates.io/crates/pre. Aufruf am 01.08.2020.

[4] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, pp. 40–51, Oct 1992.

[5] Y. Liu and H. C. Cunningham, "Software component specification using design by contract," in *Proceeding of the SouthEast Software Engineering Conference, Tennessee Valley Chapter, National Defense Industry Association*, 2002.

[6] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass — java with assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 103 – 117, 2001. RV'2001, Runtime Verification (in connection with CAV '01).

[7] R. Plosch, "Design by contract for Python," in *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, pp. 213–219, Dec 1997.

[8] T. Herzog, "'contracts' Rust Bibliothek." https://crates.io/crates/contracts. Aufruf am 31.07.2020.

[9] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A bounded verifier for Rust (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 75–80, Nov 2015.

[10] Z. HUANG, Y. J. WANG, and J. LIU, "Detecting unsafe raw pointer dereferencing behavior in Rust," *IEICE Transactions on Information and Systems*, vol. E101.D, pp. 2150–2153, Aug 2018.

[11] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the Rust programming language," *Proc. ACM Program. Lang.*, vol. 2, Dec. 2017.

[12] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for Rust," *Proc. ACM Program. Lang.*, vol. 4, Dec. 2019.

[13] T. Diekmann, "'safety-guard' Rust Bibliothek." https://crates.io/crates/safety-guard. Aufruf am 31.07.2020.

[14] M. Farrokhzad and M. Bryan, "'safe' Rust Bibliothek." https://crates.io/crates/safe. Aufruf am 31.07.2020.

[15] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, pp. 42–51, Jan 2002.

[16] M. Das, M. A. Fahndrich, R. Venkatapathy, D. W. Weise, W. H. Hudson, S. H. Agarwal, W. H. Shihara, H. Ruescher, S. W. Low, F. S. Terek, *et al.*, "Source code annotation language," Sept. 1 2009. US Patent 7,584,458.

[17] "Using SAL annotations to reduce C/C++ code defects." https://docs.microsoft.com/en-us/cpp/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects?view=vs-2019. Aufruf am 01.08.2020.

[18] "The Rust reference: Unsafety." https://doc.rust-lang.org/stable/reference/unsafety.html. Aufruf am 10.08.2020.

[19] "The Rust programming language: Appendix G - how Rust is made and „nightly Rust"." https://doc.rust-lang.org/stable/book/appendix-07-nightly-rust.html. Aufruf am 10.08.2020.

[20] "The Rust reference: Items." https://doc.rust-lang.org/reference/items.html. Aufruf am 04.08.2020.

[21] "The Rustonomicon: Zero sized types (ZSTs)." https://doc.rust-lang.org/nomicon/exotic-sizes.html#zero-sized-types-zsts. Aufruf am 01.08.2020.

[22] "The Rust reference: Implementations." https://doc.rust-lang.org/reference/items/implementations.html. Aufruf am 10.08.2020.

[23] "Safety-Abschnitt der Dokumentation, des std::ptr-moduls." https://doc.rust-lang.org/std/ptr/index.html#safety. Aufruf am 01.08.2020.

[24] "The Rust reference: Procedural macros." https://doc.rust-lang.org/reference/procedural-macros.html. Aufruf am 01.08.2020.

[25] B. Kauschke, "Tracking issue for 'min_const_generics'." https://github.com/rust-lang/rust/issues/74878. Aufruf am 01.08.2020.

[26] "The Rust reference: Identifiers." https://doc.rust-lang.org/reference/identifiers.html. Aufruf am 01.08.2020.

[27] M. Farrokhzad, "Tracking issue for RFC 2457, „allow non-ascii identifiers"." https://github.com/rust-lang/rust/issues/55467. Aufruf am 01.08.2020.

[28] D. Tolnay, "Unit struct with type parameters." https://github.com/dtolnay/case-studies/tree/master/unit-type-parameters. Aufruf am 03.08.2020.

[29] "The Rust reference: Method-call expressions." https://doc.rust-lang.org/reference/expressions/method-call-expr.html. Aufruf am 04.08.2020.

[30] "The rust programming language: Using the newtype pattern to implement external traits on external types." https://doc.rust-lang.org/book/ch19-03-advanced-traits.html#using-the-newtype-pattern-to-implement-external-traits-on-external-types. Aufruf am 15.08.2020.

[31] B. Anderson, "Tracking issue for stmt_expr_attributes: Add attributes to expressions, etc.." https://github.com/rust-lang/rust/issues/15701. Aufruf am 06.08.2020.

[32] D. Tolnay, "Rust Latam: procedural macros workshop: sorted 'match expr'." https://github.com/dtolnay/proc-macro-workshop/blob/master/sorted/tests/05-match-expr.rs. Aufruf am 06.08.2020.

[33] D. Tolnay, "visit_mut-Modul der syn-Bibliothek." https://docs.rs/syn/1.0.38/syn/visit_mut/index.html. Aufruf am 06.08.2020.

[34] D. Tolnay, "'syn' Rust Bibliothek." https://crates.io/crates/syn. Aufruf am 16.08.2020.

[35] D. Tolnay, "'quote' Rust Bibliothek." https://crates.io/crates/quote. Aufruf am 16.08.2020.

[36] CreepySkeleton, "'proc-macro-error' Rust Bibliothek." https://crates.io/crates/proc-macro-error. Aufruf am 16.08.2020.

[37] D. Tolnay, "'trybuild' Rust Bibliothek." https://crates.io/crates/trybuild. Aufruf am 16.08.2020.

[38] N. Schwarzlose, "Anfrage zur Integration von *pre* in 'secstr'." https://github.com/myfreeweb/secstr/issues/17. Aufruf am 06.08.2020.

[39] N. Schwarzlose, "Anfrage zur Integration von *pre* in 'arrayvec'." https://github.com/bluss/arrayvec/issues/162. Aufruf am 06.08.2020.

[40] N. Schwarzlose, "Anfrage zur Integration von *pre* in 'bytes'." https://github.com/tokio-rs/bytes/issues/423. Aufruf am 06.08.2020.

[41] N. Schwarzlose, "Anfrage zur Integration von *pre* in 'smallstr'." https://github.com/murarth/smallstr/issues/9. Aufruf am 06.08.2020.

[42] N. Schwarzlose, "Anfrage zur Integration von *pre* in 'smallvec'." https://github.com/servo/rust-smallvec/issues/124#issuecomment-661939576. Aufruf am 06.08.2020.

[43] bluss, "Regelung zur Aufnahme von Abhängigkeiten in 'arrayvec'." https://github.com/bluss/arrayvec/pull/123#issuecomment-526904486. Aufruf am 07.08.2020.

[44] "The Rust reference: Conditional compilation." https://doc.rust-lang.org/reference/conditional-compilation.html#the-cfg_attr-attribute. Aufruf am 06.08.2020.

[45] G. V, "'secstr' Rust Bibliothek." https://crates.io/crates/secstr. Aufruf am 07.08.2020.

[46] N. Schwarzlose, "Fix 'safety' docs for 'from_raw_parts_mut'." https://github.com/rust-lang/rust/pull/74450. Aufruf am 07.08.2020.

[47] N. Schwarzlose, "Apply #66379 to '*mut t' 'as_ref'." https://github.com/rust-lang/rust/pull/74568. Aufruf am 07.08.2020.

[48] N. Schwarzlose, "Improve documentation of 'String::from_raw_parts'." https://github.com/rust-lang/rust/pull/74741. Aufruf am 07.08.2020.

[49] N. Schwarzlose, "Add safety section to 'NonNull::as_*' method docs." https://github.com/rust-lang/rust/pull/75266. Aufruf am 07.08.2020.

[50] N. Schwarzlose, "The 'partialeq' implementations of 'secvec<t>' and 'secbox<t>' are ub (and wrong) in some cases." https://github.com/myfreeweb/secstr/issues/20. Aufruf am 14.08.2020.

[51] scottmcm, "Isn't a pointer cast just a more dangerous transmute?." https://users.rust-lang.org/t/isnt-a-pointer-cast-just-a-more-dangerous-transmute/47007/8. Aufruf am 14.08.2020.

[52] N. Schwarzlose, "Fix some undefined behavior and check preconditions of 'unsafe' code." https://github.com/myfreeweb/secstr/pull/18. Aufruf am 15.08.2020.

[53] "Contract programming - D programming language." https://dlang.org/spec/contracts.html. Aufruf am 08.08.2020.

[54] A. Blanchard, "Introduction to C program proof with Frama-C and its WP plugin." https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf. Aufruf am 08.08.2020.

[55] Tylerlee12, "Kommentar zu 'pre: a crate to offer compile-time assistance for working with unsafe code' auf reddit." https://www.reddit.com/r/rust/comments/hrixq7/pre_a_crate_to_offer_compiletime_assistance_for/fy5uq0l/?utm_source=share&utm_medium=web2x, 15.07.2020. Aufruf am 08.08.2020.

[56] ICosplayLinkNotZelda, "Kommentar zu 'pre: a crate to offer compile-time assistance for working with unsafe code' auf reddit." https://www.reddit.com/r/rust/comments/hrixq7/pre_a_crate_to_offer_compiletime_assistance_for/fy4vjzc/?utm_source=share&utm_medium=web2x, 15.07.2020. Aufruf am 08.08.2020.

[57] J. Gjengset, "'bus' Rust Bibliothek." https://crates.io/crates/bus. Aufruf am 08.08.2020.

A Anhang: Codegenerierungsbeispiele

```
use pre::pre;

#[pre("zeichenkettenvorbedingung")]
#[pre(valid_ptr(zeiger, r+w))]
#[pre(proper_align(zeiger))]
#[pre(!zeiger.is_null())]
fn beispielfunktion(zeiger: *const u8) {}

#[pre]
fn main() {
    #[assure(
        "zeichenkettenvorbedingung",
        reason = "hier wuerde ein Grund stehen"
    )]
    #[assure(
        valid_ptr(zeiger, r+w),
        reason = "hier wuerde ein Grund stehen"
    )]
    #[assure(
        proper_align(zeiger),
        reason = "hier wuerde ein Grund stehen"
    )]
    #[assure(
        !zeiger.is_null(),
        reason = "hier wuerde ein Grund stehen"
    )]
    beispielfunktion(&0);
}
```

Abbildung 8: Beispielcode, um die Codegenerierung zu demonstrieren.

```rust
use pre::pre;

// Dokumentation gekuerzt.
#[doc = "..."]
fn beispielfunktion(
    zeiger: *const u8,
    #[cfg(all(not(doc),))] _: (
        ::pre::ValidPtrCondition<"zeiger", "r+w">,
        ::pre::ProperAlignCondition<"zeiger">,
        ::pre::BooleanCondition<"! zeiger . is_null()">,
        ::pre::CustomCondition<"zeichenkettenvorbedingung">,
    ),
) {
    ::core::debug_assert!(
        !zeiger.is_null(),
        "boolean precondition was wrongly assured: `{}`",
        ::core::stringify!(!zeiger.is_null())
    );
}

fn main() {
    beispielfunktion(
        &0,
        #[cfg(all(not(doc),))]
        (
            ::pre::ValidPtrCondition::<"zeiger", "r+w">,
            ::pre::ProperAlignCondition::<"zeiger">,
            ::pre::BooleanCondition::<"! zeiger . is_null()"
                >,
            ::pre::CustomCondition::<"
                zeichenkettenvorbedingung">,
        ),
    );
}
```

Abbildung 9: Aus Abbildung 8 mit dem „nightly“ Compiler generierter Code. Die gekürzte Dokumentation ist in Abbildung 11 zu finden.

```
use pre::pre;

// Dokumentation gekuerzt.
#[allow(non_camel_case_types)]
#[allow(non_snake_case)]
#[cfg(all(not(doc),))]
struct beispielfunktion {
    _custom_zeichenkettenvorbedingung: (),
    _valid_ptr_zeiger_rw: (),
    _proper_align_zeiger: (),
    _boolean__21_20zeiger_20_2e_20is__null_28_29: (),
}

#[doc = "..."]
fn beispielfunktion(
    zeiger: *const u8,
    #[cfg(all(not(doc),))] _: beispielfunktion,
) {
    ::core::debug_assert!(
        !zeiger.is_null(),
        "boolean precondition was wrongly assured: `{}`",
        ::core::stringify!(!zeiger.is_null())
    );
}
fn main() {
    beispielfunktion(
        &0,
        #[cfg(all(not(doc),))]
        beispielfunktion {
            _custom_zeichenkettenvorbedingung: (),
            _valid_ptr_zeiger_rw: (),
            _proper_align_zeiger: (),
            _boolean__21_20zeiger_20_2e_20is__null_28_29: ()
                ,
        },
    );
}
```

Abbildung 10: Aus Abbildung 8 mit dem „stable“ Compiler generierter Code. Die gekürzte Dokumentation ist in Abbildung 11 zu finden.

```
# This function has preconditions
This function has the following preconditions generated by
   [‘pre‘ attributes](https://docs.rs/pre/0.2.0/pre/attr.pre
   .html):

- zeichenkettenvorbedingung
- the pointer ‘zeiger‘ must be valid for reads and writes
- the pointer ‘zeiger‘ must have a proper alignment for its
   type
- ‘! zeiger . is_null()‘

To call the function you need to [‘assure‘](https://docs.rs/
   pre/0.2.0/pre/attr.assure.html) that the preconditions
   hold:

‘‘‘rust,ignore
#[assure(
   "zeichenkettenvorbedingung",
    reason = "<specify the reason why you can assure this
       here>"
)]
#[assure(
    valid_ptr(zeiger, r+w),
    reason = "<specify the reason why you can assure this
       here>"
)]
#[assure(
    proper_align(zeiger),
    reason = "<specify the reason why you can assure this
       here>"
)]
#[assure(
    ! zeiger . is_null(),
    reason = "<specify the reason why you can assure this
       here>"
)]
beispielfunktion(/* parameters omitted */);
‘‘‘
```

Abbildung 11: Die Abbildung 8 generierte Dokumentation für die Funktion `beispielfunktion`, die in Abbildung 9 und 10 gekürzt wurde.

```
#[pre::extern_crate(std)]
pub mod pre_std {
    mod ptr {
        impl<T> NonNull<T>
        where
            T: ?Sized,
        {
            #[pre(!ptr.is_null())]
            const unsafe fn new_unchecked(ptr: *mut T) ->
               NonNull<T>;
        }

        #[pre(valid_ptr(dst, w))]
        #[pre(proper_align(dst))]
        unsafe fn write<T>(dst: *mut T, src: T);
    }
}
```

Abbildung 12: Beispielcode, um die Codegenerierung von `extern_crate` Modulen zu demonstrieren.

```rust
// Dokumentation gekuerzt (1)
#[doc = "..."]
pub mod pre_std {
    #[allow(unused_imports)]
    use pre::pre;

    #[allow(unused_imports)]
    #[doc(no_inline)]
    pub use std::*;

    // Dokumentation gekuerzt (2)
    #[doc = "..."]
    pub mod ptr {
        #[allow(unused_imports)]
        use pre::pre;

        #[allow(unused_imports)]
        #[doc(no_inline)]
        pub use std::ptr::*;

        // Dokumentation gekuerzt (4)
        #[doc = "..."]
        #[pre(! ptr . is_null())]
        // Verhindert, dass Dokumentation erneut generiert
           wird
        #[pre(no_doc)]
        #[pre(no_debug_assert)]
        #[inline(always)]
        #[allow(non_snake_case)]
        pub fn NonNull__impl__new_unchecked__() {}

        #[pre(valid_ptr(dst, w))]
        #[pre(proper_align(dst))]
        // Dokumentation gekuerzt (3)
        #[doc = "..."]
        #[inline(always)]
        pub unsafe fn write<T>(dst: *mut T, src: T) {
            std::ptr::write(dst, src)
        }
    }
}
```

Abbildung 13: Der aus Abbildung 12 mit dem „nightly“ Compiler generierte Code. Der mit dem „stable“ Compiler generierte Code unterscheidet sich nur geringfügig durch die Dokumentation. Die gekürzte Dokumentation ist in den Abbildungen 14, 15, 16 und 17 zu finden.

```
[‘pre‘ definitions](https://docs.rs/pre/0.2.0/pre/attr.pre.
    html) for the [‘std‘](module@std) crate.

This module was generated by a [‘extern_crate‘ attribute](
    https://docs.rs/pre/0.2.0/pre/attr.extern_crate.html).
It acts as a drop-in replacement for the ‘std‘ module.
```

Abbildung 14: Die aus Abbildung 13 gekürzte Dokumentation (1).

```
[‘pre‘ definitions](https://docs.rs/pre/0.2.0/pre/attr.pre.
    html) for the [‘std::ptr‘](module@std::ptr) module.

This module was generated by a [‘extern_crate‘ attribute](
    https://docs.rs/pre/0.2.0/pre/attr.extern_crate.html).
It acts as a drop-in replacement for the ‘std::ptr‘ module.
```

Abbildung 15: Die aus Abbildung 13 gekürzte Dokumentation (2).

```
[‘std::ptr::write‘](value@std::ptr::write) with
    preconditions.

This function behaves exactly like ‘std::ptr::write‘, but
    also has preconditions checked by ‘pre‘.

**You should also read the [Safety section on the
    documentation of ‘std::ptr::write‘](value@std::ptr::write
    #safety).**
```

Abbildung 16: Die aus Abbildung 13 gekürzte Dokumentation (3).

```
A stub for the preconditions of the [`std::ptr::NonNull<T>::
    new_unchecked`](value@std::ptr::NonNull::new_unchecked)
    function.

# What is this function?

This function was generated by an `impl` block inside a [`
    extern_crate` attribute](https://docs.rs/pre/0.2.0/pre/
    attr.extern_crate.html) that looked like this:

```rust,ignore
impl< T > NonNull < T > where T : ? Sized, {
    const unsafe fn new_unchecked(ptr : * mut T) -> NonNull
        < T >;
}
```

Preconditions on external functions inside of an `impl`
    block are attached to empty functions like this one.
When the preconditions should be checked, a call to this
    function is inserted, which triggers checking the
    preconditions.

This function has preconditions

This function has the following precondition generated by
    the [`pre` attribute](https://docs.rs/pre/0.2.0/pre/attr.
    pre.html):

- `! ptr . is_null()`

To call the function you need to [`assure`](https://docs.rs/
    pre/0.2.0/pre/attr.assure.html) that the precondition
    holds:

```rust,ignore
#[forward(impl test_pre::pre_std::ptr::NonNull)]
#[assure(
    ! ptr . is_null(),
    reason = "<specify the reason why you can assure this
        here>"
)]
new_unchecked(/* parameters omitted */);
```
```

Abbildung 17: Die aus Abbildung 13 gekürzte Dokumentation (4).

```rust
#[pre::pre]
fn main() {
    unsafe {
        #[forward(impl pre_std::ptr::NonNull)]
        #[assure(
            !ptr.is_null(),
            reason = "eine Referenz ist nie null"
        )]
        std::ptr::NonNull::new_unchecked(&mut 0);
    }
}
```

Abbildung 18: Code, der die in Abbildung 12 definierten Vorbedingungen für die `NonNull::new_unchecked` Funktion nutzt.

```rust
fn main() {
    unsafe {
        if true {
            std::ptr::NonNull::new_unchecked(&mut 0)
        } else {
            pre_std::ptr::NonNull__impl__new_unchecked__(
                #[cfg(all(not(doc),))]
                (::pre::BooleanCondition::<"! ptr . is_null
                    ()">,),
            );
            unreachable!()
        };
    }
}
```

Abbildung 19: Der aus Abbildung 18 mit dem „nightly“ Compiler generierte Code. Ein Methodenaufruf würde analog generiert werden.

```
fn main() {
    unsafe {
        if true {
            std::ptr::NonNull::new_unchecked(&mut 0)
        } else {
            pre_std::ptr::NonNull__impl__new_unchecked__(
                #[cfg(all(not(doc),))]
                pre_std::ptr::NonNull__impl__new_unchecked__
                    {
                    _boolean__21_20ptr_20_2e_20is__null_28_29
                        : (),
                },
            );
            unreachable!()
        };
    }
}
```

Abbildung 20: Der aus Abbildung 18 mit dem „stable“ Compiler generierte Code. Ein Methodenaufruf würde analog generiert werden.