Assertion-based Testing of Go Programs

Eva Charlotte Mayer

Department of Informatics
Software and Systems Engineering

Master's Thesis in Informatics

Assertion-based Testing of Go Programs

Eva Charlotte Mayer

Supervisors Prof. Dr. Alexander Pretschner
Software and Systems Engineering
Department of Informatics
Technical University Munich, Germany

Prof. Dr. Peter Müller
Programming Methodology Group
Department of Computer Science
ETH Zurich, Switzerland

Advisors M.Sc. Linard Arquint and M.Sc. Felix Wolf
Programming Methodology Group
Department of Computer Science
ETH Zurich, Switzerland

Eva Charlotte Mayer
Assertion-based Testing of Go Programs
Testen von Go Programmen mit Assertions
Master's Thesis in Informatics
Supervisors: Prof. Dr. Alexander Pretschner and Prof. Dr. Peter Müller
Advisors: M.Sc. Linard Arquint and M.Sc. Felix Wolf
Submission: November 8, 2020

Technical University Munich
Software and Systems Engineering
Department of Informatics
Boltzmannstraße 3
85748 Garching bei München, Germany

I confirm that this Master's thesis is my own work and I have documented all sources and material used.

Eva Charlotte Mayer
Zurich, Switzerland. November 8, 2020.

Für Claire und Herbert Klopries.

Abstract

Runtime checking based on assertions can identify erroneous executions of a program. It is thus complementary to static verification that proves the absence of errors. We develop an assertion-based testing framework for the Go programming language. Our work comprises the design of a specification language that takes Go idiosyncrasies into consideration. We further develop a tool to generate runtime checks for the specification annotations. The tool is enhanced with a prototype for test input generation which allows for testing programs against a specification. Our work concludes with an extensive evaluation on the performance and effectiveness of the implemented framework. The evaluation shows that the assertion-based testing framework is a useful supplement to the Go verifier Gobra. Since both tools use a similar specification syntax, we can reuse annotations from runtime checking for verification. Thus, the thesis contributes toward both assuring program correctness and bridging the gap between runtime checking and verification.

Acknowledgements

I would like to thank Prof. Dr. Alexander Pretschner and Prof. Dr. Peter Müller for making the Master's thesis at the ETH Zurich possible. I am grateful for the assistance given by my two advisors Linard Arquint and Felix Wolf. It was a pleasure to work and learn from both of you and I highly enjoyed receiving both your helpful advice and challenging questions. I would also like to express gratitude to the whole Programming Methodology group, especially Sandra Schneider, for making me feel welcome right from the start.

Special thanks go to my family and friends, above all Susanne Klopries-Mayer, Benjamin Mayer, Harald Brandenburg, Dominik Widmann, and the Hessis for proof-reading my thesis and their support throughout my Master's studies.

Finally, I would like to express my appreciation to the Stiftung der Deutschen Wirtschaft for the financial and idea contributions towards my education.

Contents

1 Introduction

1.1 Motivation

Runtime assertions are useful for identifying bugs in programs and thus increase confidence in the correctness of a program [25, 44]. Runtime assertions are predicates that reason about a given state of the program at some point of the execution [8]. If an assertion does not hold, the program is deemed to be in a faulty state. In comparison to unit tests that consider only function outputs, assertion-based tests allow for checks on arbitrary points in the program [23]. In the right circumstances, this enables a *programmer*, i.e. a person implementing a program, to find the sources of errors faster than with unit tests.

Consider the program given in Figure 1.1. The function `binarySearch` is intended to decide whether a given value `s` exists in a slice. In Go, a slice is a dynamically-sized view at a particular offset into an underlying array [14]. The function returns the position `pos` of the value, or -1 if the value is not in the slice. The binary search algorithm relies on the fact that the input slice is sorted in increasing order [26]. With sorted input, the function works as exptected: Calling `binarySearch([]int{1,2,3,4}, 3)` returns 2 and calling `binarySearch([]int{}, 42)` or `binarySearch([]int{1,2,3,4}, 42)` both correctly return -1. However,

```
1  func binarySearch(s []int, x int) (pos int) {
2      low := 0
3      high := len(s)
4      pos = -1
5
6      for low < high {
7          mid := (low + high) / 2
8          if s[mid] == x {
9              pos = mid
10             break
11         } else if s[mid] < x {
12             low = mid + 1
13         } else {
14             high = mid - 1
15         }
16     }
17
18     return
19 }
```

Fig. 1.1: Binary search [26] implementation in Go

when the function is called with an slice that is not sorted in increasing order, the algorithm returns faulty results. E.g. `binarySearch([]int{4,3,2,1}, 3)` returns -1 even though 3 is in the slice at position 1. The requirement, that the function should only be used with sorted slices, is not checked. Thus, nothing prevents a *client* of the function, i.e. a person that uses the function, from calling it with malformed input.

Established software engineering principles advocate that a *client* of a function, i.e. a person that uses the function, should not need to worry about an implementation but should be able to fully rely on the function's signature and documentation [41]. As documentation might not exist or be separated from the code, it can easily happen that a client violates certain properties when calling a function. Formal *specification* tries to address this issue by avoiding ambiguities in natural language and enabling conformity checking against the specification at the call site [19]. In this thesis, specification is given in form of runtime assertions. This specification can be used for runtime assertion checking or verification. It could also simply serve as a more precise description of the program opposed to using natural language.

Figure 1.2 has added specification to the implementation of the `binarySearch` function. The specification states that the input slice `s` for the binary search function

```
1  //@ requires forall i, j int :: 0 <= i < len(s) && 0 <= j < i
2  //@                             ==> s[i] <= s[j]
3  //@ ensures 0 <= pos && pos < len(s) && s[pos] == x
4  //@         || pos == -1 && ! (exists i int :: i in range s && s[i] == x)
5  func binarySearch(s []int, x int) (pos int) {
6      low := 0
7      high := len(s)
8      pos = -1
9
10     //@ invariant 0 <= low && low <= high && high <= len(s)
11     for low < high {
12         mid := (low + high) / 2
13         if s[mid] == x {
14             pos = mid
15             break
16         } else if s[mid] < x {
17             low = mid + 1
18             //@ assert forall i int :: 0 <= i < low ==> s[i] != x
19         } else {
20             high = mid - 1
21             //@ assert forall i int :: high < i < len(s) ==> s[i] != x
22         }
23     }
24
25     return
26 }
```

Fig. 1.2: Binary search [26] implementation annotated with runtime assertions

has to be sorted in increasing order (lines 1-2). For the output of the function is is ensured that either the position of value `x` is returned, or -1 if the value `x` does not exist in slice `s` (lines 3-4). In addition to the specification given as part of the function signature, the function's body includes specification as well: It is specified that the helper variables `low` and `high` are always valid indices for the slice before and after each loop iteration (line 10). Additionally, whenever `low` is adjusted, no value left of `low`, precisely no value at a strictly smaller index than `low`, is equal to the value we search for (line 18). Similarly, whenever `high` is adjusted, no value right of `high`, i.e. no value at strictly greater index than `high`, is equal to `x`. The runtime checks generated for these assertions catch invalid inputs and report them to the user. Thus, the runtime assertions warrant safe usage and correct results of the function provided that the specification is correct and precise.

1.2 Supplementing Verification

The verification tool Gobra [2] proves correctness of programs written in Go [18]. Verification requires very thorough specifications to prove properties that hold for all inputs, thread schedules, and program environments. Thus, specification for verification is hard to write for programmers without experience in formal methods [19]. Nevertheless, we can facilitate verification of Go programs with the aid of runtime assertions:

Runtime assertions are typically easier to formulate as they reason about specific runtime values and also give weaker guarantees by only specifying certain properties. E.g. permissions are normally not used and also aliasing is often not expressed [23]. When invoking a program with different inputs, successful runtime assertion checks guarantee that the specified properties hold for exactly these program executions. The guarantees do not imply correctness of the program when run with other inputs or in different environments. In this regard, runtime assertion checking is similar to unit testing. Even though runtime assertions do not provide as strong guarantees for program correctness as verification, they can still provide a solid basis of specification that can be employed for verification [7].

Typically, a verification is performed after the implementation of a program. In addition, verification is often attempted as a collective effort of multiple people. The process of iteratively writing specification and attempting a verification can be accelerated if the code base provides clear assumptions and guarantees about a program's behavior that can be transformed into specification. Hence, a programmer should be encouraged to explicitly state these assumptions and guarantees when implementing a program, e.g. by means of runtime assertions. Runtime assertions

give a programmer immediate benefit by enabling a programmer to test conformity of the program with the assumptions and guarantees that were made.

Furthermore, runtime assertion checks facilitate validation of a specification: an unsuccessful assertion check at runtime provides an immediate counter example, which can be used to inspect the specification [31]. Hence, rigorous runtime checking before applying verification not only decreases the likelihood of bugs in the program but also gives programmers confidence in the specification. Thus, we conclude that runtime checking can supplement verification [19, 44].

1.3 Contributions

In our work we develop an assertion-based runtime checking framework for Go programs which supplements verification with Gobra. Specifically, we solve the following challenges:

- Development of a specification language for runtime assertion checking based on the Gobra specification syntax.
- Runtime check generation for specification annotations such that assertions specified in a program are checked while executing the program.
- Evaluation of the performance of the assertion-based runtime checking framework and its effectiveness in assisting verification with Gobra.
- Test input generation to check a program against its specification using different inputs.

In summary, the goal of this thesis is to provide a runtime assertion checking framework, which motivates Go programmers to write specification that serves as a basis for verification with Gobra. The thesis thus contributes to bridging the gap between programming and verification.

The framework that is developed during this thesis is called GoRAC; Go Runtime Assertion Checker. The thesis is structured as follows: Related work is presented in Chapter 2. Chapter 3 describes the specification languages of GoRAC. The translation of the specification language to runtime checks is detailed in Chapter 4. The implementation of GoRAC is discussed in Chapter 5. Chapter 7 evaluates GoRACin terms of performance, expressiveness of its language, and its potential for verification. Test input generation for GoRAC, taking certain specification into account, is explained in Chapter 6. Chapter 8 concludes the thesis and gives an outlook for potential future work.

2 Related Work

Native vs. third party runtime assertion checking. Various programming languages allow for runtime assertion checking. While some programming languages natively support assertions as part of their syntax, others can be checked using third party tools. Eiffel [29] is an object-oriented programming language which natively supports the concept of design-by-contract through invariants, pre- and postconditions. Similar concepts can be found in Spec# [4]. The Spec# programming system extends C# with a specification language and includes several features such as a compiler which can generate runtime checks for specification annotations, or a programming methodology that gives rules for structuring programs and for using specifications [4]. Ada [38] and SPARK [3], which defines a subset of the Ada programming language, also have built-in language support for design-by-contract. Ada is designed to improve code safety: It has a strong type system, supports compile-time checks based on e.g. named closing of blocks, and also enables runtime checking to prevent buffer overflows or range violations. Third party tools for runtime assertion checking exist for Java: Jahob [44] is a verification system for programs written in a subset of Java which i.a. can be used for runtime checking. The Java Modeling Language (JML) [23] is a specification language for Java programs. Besides a runtime assertion checker for JML [24], several other tools build on JML annotations such as a unit test generator [45] which generates JUnit test code from JML annotations. C# and other .NET languages integrate Code Contracts [13] for runtime assertion checking. A third party runtime checker for C++ is Boot.Contract [9]. Furthermore, [34] details runtime checking of pre- and postconditions for Scala.

Synergies between runtime checking and verification. Various work has explored how runtime checking can supplement verification [7, 19]. Many programming languages can be annotated with the same specification language for both runtime assertion checking and verification. For Java, both the jmlc compiler [24] and the Extended Static Checker (ESC/Java2) [11] are based on JML. The jmlc compiler translates JML assertions into runtime checks while ESC/Java2 verifies a program against its JML specification. The Jahob verification system provides both a verifier and a runtime assertion checker for its subset of Java. Likewise, Spark supports both static and dynamic verification [3]. Besides the compiler that generates runtime checks from annotations, the Spec# programming system also includes a static

program verifier [4]. Moreover, efforts have been made to supply programmers with meaningful counter examples when verification using Spec# fails [31].

Runtime assertion checking for Go. To the best of our knowledge no comprehensive assertion package for Go exists that is compatible with the expressive specifications of Gobra. Existing testing tools for Go are not compatible with Gobra because the assertion language typically used by runtime checkers is less expressive than the language used by deductive verifiers. The packages go-contracts [6], gocontracts [39] and godbc [1] provide only pre- and postcondition checks, the package dbc [42] further includes invariants, assertions and assumptions. No package supports advanced specification constructs such as quantifiers, old expressions, predicates, framing or purity assertions. The packages expect the specification to be written as Go code, either by passing the specified properties directly to Go functions that perform the checks [1, 6, 42] or as comments in Go syntax [39]. Hence none of these Go packages can be used as an assertion-based testing supplement to the Gobra verifier.

Comparison to our work. The assertion-based testing tool GoRAC that is implemented as part of this thesis supports specification constructs such as quantifiers and old expressions which are common features of runtime checkers [23, 44]. Quantifiers are used to reason about a range of elements. In GoRAC, only bounded quantifiers are supported. To the best of our knowledge, there exists no runtime assertion checker without this limitation. GoRAC includes certain optimizations for the runtime checks of quantifiers. Similar optimizations can be found for the runtime checker of the Jahob verification system [44]. Old expressions are used to reason about previous program states. GoRAC distinguishes whether an old expression includes a variable that is on the heap or not, and handles the old expression correspondingly. This distinction is not made for other tools such as JML [23] or Jahob [44].

3 Specification Language

> *Excellence isn't about meeting the spec, it's about setting the spec.*
>
> **— Seth Godin**
> (Author and Dot-Com Business Executive)

This chapter describes the specification language of the assertion-based testing framework GoRAC that was developed during this thesis. The objective of this chapter is to enable readers to write specification in form of GoRAC annotations on their own.

GoRAC annotations are stated as comments inside a Go program. GoRAC's annotation syntax was influenced by established syntax in e.g. VeriFast [21] or Javadoc tags [30]. Annotions are either prefixed by `//@` for a single line annotation or surrounded by `/*@ ... */` for multi-line annotations. This syntax coincides with Go's syntax for comments, such that a GoRAC annotation is always also a Go comment. As a consequence, GoRAC can be used together with existing development tools for Go. This is crucial for the application of GoRAC in a real-world setting.

Single line GoRAC annotations can contain multiple *specification clauses*. Likewise, multi-line specification annotations can contain multiple specification clauses that can also be split over several lines. Figure 3.1 exemplifies the different kinds of specification annotations: An annotation consisting of the specification clauses `requires x > 0 && x < 42` and `ensures x != 0` is first written as a single line comment, then as multiple consecutive single line comments, and then as a multi-line comment.

```
//@ requires x > 0 && x < 42 ensures x != 0

//@ requires x > 0 &&
//@          x < 42 ensures x != 0

/*@ requires x > 0
*    && x < 42 ensures
*    x != 0
*/
```

Fig. 3.1: Single- and multi-line specification annotations

The goal of this chapter is to introduce GoRAC's annotation syntax. Section 3.1 gives an overview of the different *specification constructs* that define the GoRAC specification language. In the remaining sections, we explain some of the specification constructs in more depths.

3.1 Syntax

In order to declare the syntax used for specification in GoRAC, we differentiate between specification clauses $\langle s \rangle$ and auxiliary declarations used in the specification $\langle d \rangle$. Furthermore, we use $\langle a \rangle$, $\langle e \rangle$, $\langle l \rangle$, $\langle x \rangle$, $\langle T \rangle$, $\langle q \rangle$, $\langle p \rangle$ and $\langle L \rangle$ to refer to assertions, expressions, literals and constants, variables, types, quantifiers, predicates and label names, respectively. The following four rules define a left-recursive grammar for the specification syntax recognized by GoRAC:

$\langle s \rangle$ ::= `assert` $\langle a \rangle$ | `assume` $\langle a \rangle$ | `requires` $\langle a \rangle$ | `ensures` $\langle a \rangle$ | `invariant` $\langle a \rangle$

GoRAC supports five different specification clauses. A specification clause consists of a keyword and an assertion $\langle a \rangle$. The keyword determines the program state in which the condition expressed by the assertion holds. We elaborate further on specification clauses in Section 3.2. The following rules explain the syntax of assertions.

$\langle a \rangle$ ::= $\langle e \rangle$ | `(`$\langle a \rangle$`)` | `!`$\langle a \rangle$ | $\langle a \rangle$ `&&` $\langle a \rangle$ | $\langle a \rangle$ `||` $\langle a \rangle$

The assertion that is part of a specification clause can be an expressions $\langle e \rangle$, a negated assertion, or a conjunction or disjunction of assertions. In the next rule, we specify the syntax of expressions that are part of the Go language [14] and can be used in GoRAC's annotations.

$\langle e \rangle$ ::= $\langle l \rangle$ | $\langle x \rangle$
| `*`$\langle e \rangle$ | `+`$\langle e \rangle$ | `-`$\langle e \rangle$ | `!`$\langle e \rangle$
| $\langle e \rangle$ `*` $\langle e \rangle$ | $\langle e \rangle$ `/` $\langle e \rangle$ | $\langle e \rangle$ `%` $\langle e \rangle$ | $\langle e \rangle$ `+` $\langle e \rangle$ | $\langle e \rangle$ `-` $\langle e \rangle$
| $\langle e \rangle$ `<` $\langle e \rangle$ | $\langle e \rangle$ `<=` $\langle e \rangle$ | $\langle e \rangle$ `>` $\langle e \rangle$ | $\langle e \rangle$ `>=` $\langle e \rangle$ | $\langle e \rangle$ `==` $\langle e \rangle$ | $\langle e \rangle$ `!=` $\langle e \rangle$
| $\langle e \rangle$ `&&` $\langle e \rangle$ | $\langle e \rangle$ `||` $\langle e \rangle$
| $\langle e \rangle$`[`$\langle e \rangle$`]` | $\langle e \rangle$`.`$\langle e \rangle$ | $\langle T \rangle$`{`$\langle e \rangle$`*}` | $\langle e \rangle$`(`$\langle e \rangle$`*)`

Goracs expressions includes most of Go's expressions, namely literals, constants, and variables (line 1); arithmetic operations (line 2,3); comparison operators (line 4); boolean operations (line 5); and finally index, dot, and call expressions, and composite literals (line 6). For readers that are not familiar with the Go expressions from line 6, we quickly illustrate their use with the following examples:

- Index expressions ⟨*e*⟩`[`⟨*e*⟩`]`: For an array or slice `a`, a valid index expression is `a[42]`. For a map `m`, an example of an index expression is `m["key"]`.
- Dot expressions ⟨*e*⟩`.`⟨*e*⟩: For a struct `s`, a dot expression `s.f` denotes an access to some field or function `f` with `s` as receiver of `f`. Dot expressions can also be used for package accesses, e.g. `package.instance`.
- Call expressions ⟨*e*⟩`(`⟨*e*⟩*`)`: Assuming some function `fooFunc(x int)` exists, then the expression `fooFunc(1337)` is a call expression.
- Composite literals ⟨*T*⟩`{`⟨*e*⟩*`}` : For instance, `foobar: 42` defines a struct literal of type `foo` whose field `bar` is set to 42. Another example is the array literal `[]int1,3,3,7`.

Note that call expressions are permitted only if the function called satisfies additional constraints discussed in Section 3.7. Moreover, a call expression can also be used to declare a predicate call. Predicate calls are no Go expressions. Their use is detailed in Section 3.6. The next rules detail further assertions and expressions that are specific to GoRAC and not part of the regular Go syntax:

⟨*a*⟩ ::= ⟨q_u⟩ | `acc(`⟨*e*⟩`)`

GoRAC introduces further constructs that aid with specifying the behavior of code: Additional to the assertions stated above, universal quantifiers ⟨q_u⟩ facilitate specifying properties of elements in a data structure. Their exact syntax is described in Section 3.3. Assertions can also be access permissions `acc(`⟨*e*⟩`)` that allow reasoning about accessible heap locations. These permissions are addressed in more detail in Section 3.5.

⟨*e*⟩ ::= `old(`⟨*e*⟩`)` | `old`⟨*L*⟩`(`⟨*e*⟩`)`
| ⟨*e*⟩ `?` ⟨*e*⟩ `:` ⟨*e*⟩

Additional to Go's expressions, GoRAC also supports old expressions (line 1) that can capture program states at previous program points. The exact syntax and semantics of old expressions is discussed in Section 3.4. Lastly, GoRAC's specification language includes the ternary operator (line 2) that enables writing conditionals within an expression. Unlike in other programming languages, the ternary operator is not supported by Go.

The above rules describe specification clauses and their assertions. Besides these specification constructs, GoRAC also permits auxiliary declarations:

⟨*d*⟩ ::= `pure` | ⟨*p*⟩ | ⟨*L*⟩`:` | `shared:` ⟨*x*⟩* | `exclusive:` ⟨*x*⟩*

Auxiliary declarations can consist of the keyword `pure` whose usage is explained in Section 3.7, or definitions of parameterized assertions $\langle p \rangle$ detailed in Section 3.6. Furthermore, GoRAC provides an annotation $\langle L \rangle$ to label program points. Labels inside specification are necessary even though the Go language also supports the declaration of labels. However, a label that is declared and only used in specification annotations, i.e. in comments, will be deemed as unused by the Go compiler. Unused labels are not permitted in Go, hence, the need for specification labels arises. The two auxiliary declarations `shared` and `exclusive` determine that the variables $\langle x \rangle^*$ are *shared* or, respectively, *exclusive*, and therefore treated differently. This distinction is discussed in detail in Section 3.4.

As mentioned above, further sections will go into detail on the different specification constructs. The remainder of Chapter 3 has the following structure: First, we address the different specification clauses in Section 3.2. Then, we declare the exact syntax of quantifiers, old expressions and access permissions in Sections 3.3 - 3.5. The last two sections of Chapter 3 concern predicate declarations and purity annotations. As a reminder, in this chapter we focus on syntax and semantics. The generation of runtime checks is covered in Chapter 4.

3.2 Specification Clauses

Specification clauses express that certain assertions must hold at specific times during program execution. For example, a user wants to express that an assertion holds before a function execution or during each iteration of a loop. The next sections deal with the different specification clauses that GoRAC supports and explain where they should be used.

3.2.1 Assert Statements & Assumptions

An assert statement `assert` $\langle a \rangle$ states that the assertion $\langle a \rangle$ holds at the program point where the statement is placed. Assert statement can be declared at arbitrary program points inside a function. An example of an assert statement is given in the following Go code. We assert that a divisor used in a division is not equal to zero:

```
//@ assert divisor != 0
result := 42 / divisor
```

GoRAC also supports assumptions `assume` $\langle a \rangle$ that behave like assertions. The reason why seemingly redundant assumptions are supported stems from verification:

Assumptions in verification express conditions which are assumed to be true. They are not checked by the verifier but instead used to provide the verifier with more information for the proof. If an assumptions contradicts existing knowledge, a verifier enters an inconsistent state in which every property holds trivially. Since we cannot model this behavior for runtime checking, we decided to treat assumptions like assertions that check at runtime whether the assumed assertion is satisfied.

3.2.2 Preconditions & Postconditions

A precondition states assumptions about arguments of a function. Because the implementation of a function can rely on these assumptions, they should always hold when the function is called [29]. Preconditions are declared as part of a function's signature; as an annotation above a function declaration. We express preconditions in GoRAC annotations using the keyword `requires`.

A postcondition expresses guarantees about the results of a function. The caller of a function has to be able to rely on these guarantees [29]. Like preconditions, postconditions are also declared as part of a function's signature. The keyword `ensures` is reserved for postconditions.

We illustrate preconditions and postconditions in the Go code below. A precondition states that the divisor parameter of a division function cannot to be zero. A postcondition ensures that the division function returns the value of the input parameter `x` divided by the divisor.

```
//@ requires divisor != 0
//@ ensures res == x / divisor
func divide(x, divisor int) (res int) {
    return x / divisor
}
```

3.2.3 Invariants

Invariants (also called loop invariants) express conditions about the program state that hold before and after each iteration of a loop [12]. In particular, an invariant holds upon entry to and exit from a loop. Loop invariants in GoRAC are expressed using the keyword `invariant`. They need to be placed before for-loop or range declarations. An example of an invariant is given on the next page; the invariant states that the sum of two variables `i` and `j` is always equal to 9. The invariant holds because when simultaneously increasing `i` while decreasing `j`, the two variables always add up to the initial value of `j`:

```
j := 9
//@ invariant i + j == 9
for i := 0; i < 10; i++  {
    j--
}
```

3.3 Quantifier

GoRAC supports both existential and universal quantifiers. Different from Gobra, where we can express unbounded quantifiers, the quantified expressions allowed by GoRAC need to be bounded. This is due to the fact that we want to precisely guarantee that a quantified assertion holds on all instances of the quantified domain. However, checking an assertion on all instances of an unbounded domain is not possible at runtime[1]. Similar restrictions are imposed for existing runtime checking tools [44, 23]. The exact syntax of bounded quantifiers is the following:

$\langle q_u \rangle$ ::= `forall` $\langle X \rangle$ `::` $\langle D(X) \rangle$ `==>` $\langle a(X) \rangle$

$\langle q_e \rangle$ ::= `exists` $\langle X \rangle$ `::` $\langle D(X) \rangle$ `&&` $\langle e(X) \rangle$

$\langle X \rangle$ ::= $(\langle x \rangle\ \langle t \rangle)(\langle x \rangle\ \langle t \rangle)^*$

We require quantifiers to have at least one quantified variable. The quantified variables denoted by $\langle X \rangle$ are used in the body of the quantifier $\langle a(X) \rangle$, $\langle e(X) \rangle$, $\langle D(X) \rangle$. Quantifiers need to have a domain $\langle D(X) \rangle$ that expresses the bounds of the quantified variables. Universal quantifiers state that all quantified variables that satisfy the domain $\langle D(X) \rangle$ also satisfy the assertion $\langle a(X) \rangle$. Existential quantifiers state that at least one quantified variable that satisfies the domain $\langle D(X) \rangle$ also satisfies the expression $\langle e(X) \rangle$. All domains need to be stated on the left side of the implication in a universal quantifier or, respectively, the conjunction in an existential quantifier.

Bounds for quantified variables can be Go data structures like arrays, slices or maps which are always finite, or finite numerical ranges. We express the bound of a quantified variable with a *domain constraint* $\langle c(x) \rangle$. The whole domain of a quantifier is a formula of conjunctions and disjunctions of domain constraints:

$\langle D(X) \rangle$ ::= $(\langle D(X) \rangle)$
| $\langle D(X) \rangle$ `&&` $\langle D(X) \rangle$

[1] There exist imprecise techniques such as sampling for checking unbounded quantifiers at runtime. However, sampling might yield scenarios where GoRAC would deem a quantifier to hold due to a lucky choice of instantiations of the quantified variables, whereas verification with Gobra would fail on the same quantifier. Therefore, it seems to be a more sustainable approach to restrict the use of quantifiers in GoRAC to bounded ones.

```
      |  ⟨D(X)⟩ || ⟨D(X)⟩
      |  ⟨c(x)⟩
⟨c(x)⟩ ::= ⟨e⟩ < ⟨x⟩ < ⟨e⟩
      |  ⟨e⟩ <= ⟨x⟩ < ⟨e⟩
      |  ⟨e⟩ < ⟨x⟩ <= ⟨e⟩
      |  ⟨e⟩ <= ⟨x⟩ <= ⟨e⟩
      |  ⟨x⟩ in range ⟨e⟩
      |  _ , ⟨x⟩ in range ⟨e⟩
      |  ⟨x⟩ , ⟨x⟩ in range ⟨e⟩
```

The syntax of domain constraints is only allowed in domains. It is required that the domain holds constraints for each of the quantified variables. An exception is made for boolean quantified variables which are natively bound to a domain of two truth values. Thus, we can omit the domain for quantifiers that have only boolean quantified variables.

Figure 3.2 illustrates the use of quantifiers. At the top, the function `median` returns the median of a given integer slice. The input slice is required to be sorted, which is specified using a universal quantifier. For a sorted slice, the position of the median is computed by differentiating between an odd and even length of the slice. At the bottom, the function `position` returns the index at which a given value exists in the

```
/*@ requires forall i, j int :: i in range nums && 0 <= j < i
*             ==> nums[j] <= nums[i]
*/
func median(nums []int) int {
    n := len(nums)
    if n % 2 == 1 {
        return nums[(n - 1) / 2]
    } else {
        return ( nums[n / 2] + nums[(n / 2) - 1] ) / 2
    }
}
```

```
//@ requires exists k int :: _, k in range nums && k == value
func position(nums []int, value int) (pos int) {
    for p, v := range nums {
        if v == value {
            pos = p
            break
        }
    }
    return
}
```

Fig. 3.2: Examples of a universal quantifier (at the top) and an existential quantifier (at the bottom)

slice. The precondition of the function expresses that the given value is contained in the slice with the help of an existential quantifier.

3.4 Old expressions

GoRAC supports the use of old expressions [44, 23] to reason about the state of previous program points. With `old[L](e)` we refer to the value which expression `e` had at program point `L`. `L` is either a specification label or a Go label as used for gotos. The expression `old(e)` denotes the value of `e` before the execution of the function f in whose specification `old(e)` occurs. Thus `old(e)` can be interpreted as a special case of `old[L](e)`, where the label is placed right at the beginning of the body of f. We call `old[L](e)` and `old(e)` a labeled and an unlabeled old expression, respectively.

3.4.1 Semantics

The semantics we define for old expressions are motivated by old semantics of existing tools such as Dafny [27] or Viper [32]. Dafny and Viper distinguish between the heap and the variable store. In both tools, variables are saved in the variable store, i.e. variables in old expressions are always evaluated to their current value. However, these old semantics for variables do not accurately model variables in Go. In Go, variables can be on the heap, too. Thus, the Go verifier Gobra introduced the distinction between *exclusive* variables, which are not on the heap and behave like variables in Dafny and Viper, and *shared* variables, which are on the heap. This classification determines our semantics of old.

We adopt Gobra's distinction between shared and exclusive variables for GoRAC. The syntax of GoRAC's specification language detailed in Section 3.1 includes the declarations "`shared:`" and "`exclusive:`" which declare whether a variable is shared or exclusive, respectively. Every variable used in an old expression has to be annotated as either shared or exclusive. Non-annotated variables are treated as exclusive by default.

Since we want to define the semantics of old expressions in which variables are evaluated in old states of the heap, we need to (1) define the *program state*. This includes the variable store and *heap snapshot* functions for looking up values of addresses in the heap. We further need to (2) define an *evaluation function* that evaluates old expressions in a program state. Using these definitions, we can describe the semantics of old expressions. We start by introducing heap snapshots:

Definition 1. *Let V be the set of values, $A \subsetneq V$ the set of addresses, and `L` a label for some program point. Then, the mapping*

$$\hbar^L : A \rightarrow V$$

designates a snapshot of the heap which captures the state of the heap at the program point labeled with `L`.

We want to highlight that A is a subset of V, i.e. that an address is also a value. We model program states such that they include both a store for local variables and a map of labels to heap snapshots of previous program points.

Definition 2. *Let s denote a store, i.e. a map from local variables to values, and m a map of labels to heap snapshots of previous program points. Then, we define the program state ρ as a tuple of store and heap snapshot map:*

$$\rho = (s, m)$$

In addition, we define the function $s(\mathtt{x})$ to lookup the value of a local variable `x`, and $m(L) := \hbar^L$ to receive the heap snapshot at a program point with label `L`.

Now we define an evaluation function for old expressions. We can model the evaluation with a function instead of a relation, since specification annotations always behave deterministically.

Definition 3. *Let $\mathbb{E}_s$ denote the set of old expressions from the specification, P the set of program points and V the set of values. Then, we define the evaluation function*

$$eval : \mathbb{E}_s \times P \rightarrow V, \quad (\mathtt{e}, \rho) \mapsto v$$

that evaluates an old expression `e` at a given program point ρ.

Old expressions can take any pure expression as an argument. Instead of reasoning about all of these expressions individually, we want to present a first important observation. We start with the following definition:

Definition 4. *A function is called heap-independent if its evaluation does not rely on a state of the heap.*

For example, the expression `*x` is not heap-independent, i.e. *heap-dependent*, due to the fact that the dereferencing operation requires a heap lookup, whereas the expression `x == 5` is heap-independent [33].

We observe that we can exchange the evaluation order of any heap-independent function `f` with `old`. This means that the following implication holds:

$$\texttt{f heap-independent} \Rightarrow$$

$$\texttt{old[L}_0\texttt{](f(old[L}_1\texttt{](e}_1\texttt{), ..., old[L}_n\texttt{](e}_n\texttt{), e}_{1'}\texttt{, ..., e}_{m'}\texttt{))} \equiv$$
$$\texttt{f(old[L}_1\texttt{](e}_1\texttt{), ..., old[L}_n\texttt{](e}_n\texttt{), old[L}_0\texttt{](e}_{1'}\texttt{), ..., old[L}_0\texttt{](e}_{m'}\texttt{))}$$

The right side of the implication is a semantic equivalence between two expressions containing old expressions. On the left side of the equivalence, the old value of some expression that includes a function call of a heap-independent function `f` at some label $\texttt{L}_0$ is looked up. The parameters of the function call are either old expressions $\texttt{old[L}_1\texttt{](e}_1\texttt{), ..., old[L}_n\texttt{](e}_n\texttt{)}$ or regular expressions $\texttt{e}_{1'}\texttt{, ..., e}_{m'}$. On the left side of the equivalence, the evaluation of the function `f` is performed using the old values of the parameters. The values of the parameters that are not old expressions are looked up at label $\texttt{L}_0$ before being passed to the function. For a parameter that itself is an old expression at some label $\texttt{L}_i$, no further old value lookup at label $\texttt{L}_0$ is performed. This is due to the fact that for arbitrary labels `A` and `B`, we have

$$\texttt{old[A](old[B](e))} \equiv \texttt{old[B](e)}$$

Thus, the equivalence shows the possibility to exchange the evaluation order of a heap-independent function with old. Intuitively, the equivalence allows us to postpone the evaluation of old after the evaluation of a heap-independent function.

We want to underline that a lot of Go's operations satisfy heap-independence. For instance, any arithmetic operation is heap-independent: Consider the addition function with two parameters $\texttt{f} : (x, y) \mapsto x + y$, then the equality from above holds:

$$\texttt{old[L](e1 + e2)} = \texttt{old[L](e1) + old[L](e2)}$$

An example of a function where the equation does not hold is the dereferencing function because it depends on the heap.

$$\texttt{old[L](*e)} \neq \texttt{*old[L](e)}$$

This inequality arises from the fact that the object `e` points to might change in between the program point of the label `L` and the program point the old expression is evaluated at.

With the distinction between shared and exclusive variables, the introduction of an evaluation function for old expressions, and the definition of heap-independent functions, we have covered all preliminaries necessary to define the semantics of old expressions. We begin with the old semantics of variables:

Definition 5. *Let $\mathbb{X}$ be the set of variables, $eval$ the evaluation function and `old[L](x)` an old expression at program point $\rho = (m, s)$ where $m(L) = \hbar^L$. Then, we define the following old semantics for variables:*

$$eval(\texttt{old[L](x)}, \rho) := \begin{cases} \hbar^L(x) & x \text{ shared} \\ s(x) & x \text{ exclusive} \end{cases} \quad \forall\, x \in \mathbb{X}$$

The definition states that shared variables are evaluated in the heap snapshot of the program point labeled with L, i.e. they evaluate to their old value at the corresponding label. Exclusive variables are evaluated in the store of the program point where the old expression is stated, i.e. they always evaluate to their current value. An example for the semantics difference between shared and exclusive variables is given in Figure 3.3. Two variables x and y are defined that initially hold the same value 42. Variable x is shared and y is exclusive. After program point L, both values are assigned the same new value 1337. The assertion that holds afterwards, demonstrates that the old value of the shared variable x is 42 while the exclusive variable y evaluates to its current value 1337.

We continue with the old semantics of literals and constants. Literals and constants have the same values at any program point and can be seen constants as heap-independent functions with arity 0. Thus, old does not affect them:

Definition 6. *Let $\mathbb{L}$ be the set of literals and constants, $eval$ the evaluation function and `old[L](y)` an old expression at program point ρ. Then, we define the following old semantics for literals and constants:*

$$eval(\texttt{old[L](y)}, \rho) := eval(y, \rho) \quad \forall\, y \in \mathbb{L}$$

Next, dereferences and lookups in slices or maps, which can be interpreted as pointers to the underlying data structures, depend on the heap. Their evaluation is thus performed in the heap snapshot of the labeled program point. The evaluation of the old expression is relayed onto the respective data structure, i.e. the pointer, slice or map object. Hence, slices and maps evaluated to slice and map values in the old heap, respectively.

```
x, y := 42, 42 //@ shared: x exclusive: y
//@ L:
x, y := 1337, 1337
//@ assert old[L](x) == 42 && old[L](y) == 1337
```

Fig. 3.3: Specification annotations demonstrating the old semantics of shared vs. exclusive variables (The assertion on line 4 holds)

Definition 7. *Let $\mathbb{P}$ be the set of pointers, $\mathbb{A}^*$ the set of slices, $\mathbb{M}$ the set of maps, $eval$ the evaluation function, and `old[L](*e)` and `old[L](e1[e2])` old expressions at program point $\rho = (m, s)$ where $m(L) = \hbar^L$. Then, we define the following old semantics for pointers, slices and maps:*

$$eval(\texttt{old[L](*e)}, \rho) := \hbar^L(eval(\texttt{old[L](e)}, \rho)) \quad \forall\ \texttt{e} \in \mathbb{P} \tag{3.1}$$

$$eval(\texttt{old[L](e1[e2])}, \rho) :=$$
$$\hbar^L(eval(\texttt{old[L](e1)}, \rho))\ \texttt{[}eval(\texttt{old[L](e2)}, \rho)\texttt{]} \quad \forall\ \texttt{e1} \in \mathbb{A}^* \cup \mathbb{M}$$

Unlike in other programming languages where arrays are object references or act like pointers, arrays in Go are values. Therefore, array lookups are heap-independent and hence treated differently than lookups on slices or maps. Array lookups are executed on both the old values of the data structure and the index. No direct heap lookup is required for arrays since the lookup occurs when evaluating old on the array and the index. Field accesses of structs are also heap-independent and treated like arrays.

Definition 8. *Let $\mathbb{A}$ be the set of arrays, $\mathbb{S}$ the set of structs, $\mathbb{F}$ the set of struct fields, $eval$ the evaluation function, and `old[L](e1[e2])` and `old[L](e.f)` old expressions at program point ρ. Then, we define the following old semantics for arrays and structs:*

$$eval(\texttt{old[L](e1[e2])}, \rho) = eval(\texttt{old[L](e1)[}eval(\texttt{old[L](e2)}, \rho)\texttt{]}, \rho) \quad \forall\ \texttt{e1} \in \mathbb{A}$$

$$eval(\texttt{old[L](e.f)}, \rho) := eval(\texttt{old[L](e).f}, \rho) \quad \forall\ \texttt{e} \in \mathbb{S},\ \texttt{f} \in \mathbb{F}$$

All unary operations except for dereferences, whose semantic with old is defined in Equation 3.1, that are part of the GoRAC specification language, are heap-independent. Hence, an old expression containing a unary expression is evaluated on the operand of the unary expression.

Definition 9. *Let $\circ \in \{$`!`, `+`, `-`$\}$, $eval$ the evaluation function, and `old[L](`$\circ$ `e1)` an old expression at program point ρ. Then, we define the following old semantics for unary operations:*

$$eval(\texttt{old[L](}\circ\ \texttt{e1)}, \rho) = eval(\circ\ \texttt{old[L](e1)}, \rho)$$

We handle binary operations in a similar fashion since GoRAC supports only heap-independent binary expressions.

Definition 10. *Let $\circ \in \{$`+`, `-`, `*`, `%`, `\`, `>`, `<`, `>=`, `<=`, `==`, `!=`, `&&`, `||`$\}$, $eval$ the evaluation function, and `old[L](e1` $\circ$ `e2)` an old expression at program point ρ. Then, we define the following old semantics for binary operations:*

$$eval(\texttt{old[L](e1}\circ\ \texttt{e2)}, \rho) = eval(\texttt{old[L](e1)} \circ \texttt{old[L](e2)}, \rho)$$

This concludes the definition of the old semantics for all specification constructs supported by GoRAC. We continue with remarks about characteristics of old expressions that follow from the semantics. Then, Section 3.4 concludes with a short explanation on the placement of old expressions in specification annotations.

3.4.2 Remarks

The semantics of old expressions as defined above have certain implications. Since some of these implications are quite subtle, we want explicitly point them out in the following two remarks.

Syntactic sugar in Go

Go provides syntactic sugar to allow the same notation for accessing an array and a pointer to an array, and for accessing a field of a struct and a struct pointer. That means, we can abbreviate `(*arrPtr)[42]` with `arrPtr[42]`, and `(*structPtr).field` with `structPtr.field`. When using these expressions in old, it is important to differentiate whether we are dealing with an array (struct) or with a pointer to an array (struct). Figure 3.4 illustrates this problem for arrays. Both functions defined in the figure have syntactically equivalent bodies and old expressions. However, in the first case `old(a)[0]` is an access to the array in the beginning of the function, and consequently evaluates to the original value `42`. Whereas in the second case `old(a)[0]` is a lookup on the reference to the array, which evaluates to the modified value `1337`. Note that if array `a` was exclusive, `old(a)[0]` would evaluate to `1337` in both cases.

Old values of indices

For index expressions on arrays or slices, we need to pay special attention to the fact that the old value of an index is used. For instance, consider that we want to express

```
//@ require a[0] == 42
func array(a [3]int) { // shared: a
    a[0] = 1337
    //@ assert old(a)[0] == 42
}

//@ require a[0] == 42
func pointer(a *[3]int) { // shared: a
    a[0] = 1337
    //@ assert old(a)[0] == 1337
}
```

Fig. 3.4: Specification annotations demonstrating the different semantics of syntactically equivalent old expressions for array and pointer to an array

the following condition: The first value of array `a` equals the old value of the array at the current index `i`. We propose to specify this using the formulation

```
a[0] == old(a[i])
```

where variable `i` is shared. However, with this formulation, the old value of the array at the *old* index `i` would be used instead of the required `current` index. We need to make `i` exclusive or write

```
a[0] == old(a)[i]
```

For slices or maps in index expressions, if the variable referring to the respective data structure is exclusive, it is important to remember that the evaluation of the old expression on the data structure evaluates to its current value. E.g. for an old expression `s[i]` used at some program point `P` where `s` is an exclusive variable referring to some slice, we have the following evaluation:

$$\begin{aligned} \texttt{P:} \quad \texttt{old[L](s[i])} &= \hbar^{L}(\texttt{old[L](s)[old[L](i)]}) \\ &= \hbar^{L}(\underbrace{\underbrace{\quad\hbar(\texttt{s})\quad}_{\text{At program point P}}\texttt{[old[L](i)]}}_{\text{At program point L}}) \end{aligned}$$

This demonstrates that the lookup at the `i`-th slice index is performed at an earlier program point `L` than the program point `P` at which we can lookup the current value of `s`. We will discuss problems arising from such situations in Section 4.6.

3.4.3 Placement

Old expressions are allowed to be used in assertions, assumptions, invariants and postconditions. Their use is not permitted in preconditions. This is due to the fact that a precondition needs to hold before a function execution starts but old expressions always refer to program points within the execution of that function.

3.5 Permissions

In verification with Gobra, the program heap is modeled and access to it is governed by means of permissions. An access permission states that a heap location may be read or written to. Even though GoRAC does not have a similar heap model due to the significant runtime overhead it would entail, we still support access permissions so that GoRAC annotations can be more easily reused as specification for Gobra.

Access permissions can be stated for the following constructs:

- Pointers: If `p` is a pointer, e.g. a variable of type `*int`, then the expression `acc(p)` declares the access permission on `p`. Moreover, for an expression `e`, an access permission `acc(&e)` on a reference of the expression can be declared.
- Slices: If `s` is a slice, e.g. a variable of type `[]int`, then we can declare an access permission `acc(s)` for it. This grants access to all elements in the slice. (Note that in Gobra, access on each member needs to be defined separately.)
- Maps: If `m` is a map, e.g. a variable of type `map[string]bool`, then we permit access for it with `acc(m)`. As for slices, this grants access to all elements in the map.
- Indirect field accesses: Given a struct pointer `foo` that has a field named `bar`, an access permission `acc(foo.bar)` can be stated.

Note that for `acc(&e)`, it must be possible to refer to the memory address of expression `e`. Moreover, taking the address of `e` needs to be a pure operation. If `e` is a composite literal, stating a reference to it results in the allocation of the object. Taking the address of a newly allocated value is not deterministic; a different address can be returned each time the program is executed. Thus, referring to a composite literal is a non-deterministic and thereby impure operation. This restricts `acc(&e)` to be used only with expressions that are not composite literals. If an access permission is stated that does not abide by all these restrictions, the execution of GoRAC will result in an error. The functions in Figure 3.5 give examples of different access permissions:

```
//@ requires acc(X)
func add(x *int, y int) {
    *x += y
}

//@ requires acc(slice)
//@ ensures acc(slice)
func sum(slice []int) int {
    sum := 0
    for _, i := range slice {
        sum += i
    }
    return sum
}

type foo struct {
    bar int
}

//@ requires acc(foo.bar)
func setBar(f *foo, value int) {
    f.bar = value
}
```

Fig. 3.5: Examples of access permissions

Function `add` adds a value to an integer pointer. The access to the pointer is required. Function `sum` returns the sum over all members of a slice whose access is declared in the precondition. The function also transfers ownership of the slice back to the caller after termination of the function. Finally, the function `setBar` acts as a setter for the field `bar` of a `foo` struct. The access permission to the struct field is specified in the precondition of the function.

Note that in GoRAC, differently from Gobra, it is not required to declare access permissions for heap locations that are used inside functions or specification. For instance, an assertion `assert slice[i] == 42` can be checked without an access permission `acc(slice)` being given.

3.6 Predicates

GoRAC supports the use of parameterized assertions called predicates. Their use consists of the support for two separate syntax entities: predicate declarations and predicate calls. Predicate calls can occur in an assertion of a specification statement and refer to exactly one predicate declaration. The next two subsections address predicate declarations and predicate calls:

We declare predicates as part of the specification in GoRAC following the syntax

⟨*p*⟩ ::= `predicate` ⟨*P*⟩ `(`⟨*X*⟩`) {` ⟨*a*⟩ `}`

⟨*X*⟩ ::= `(`⟨*x*⟩ ⟨*t*⟩`)*`

Predicate declarations need to start with the keyword `predicate`. Predicate declarations are well-defined if they have a unique name ⟨*P*⟩ that can also not be equal to the name of a function in the program. Predicates can (but do not need to) have parameters ⟨*X*⟩ which are each defined as a tuple of a variable name and its type. The body of a predicate consists of a single assertion. Old expressions are disallowed to be used in the assertion of predicates. Since a predicate can be called in the specification of multiple functions, it would be unclear to which previous program point an old expression refers to. Additionally, predicate declarations that range over multiple lines need to be declared using multiple single line specification comments.

The GoRAC syntax described in the beginning of this chapter includes call expressions:

⟨*e*⟩ ::= ⟨*e*⟩`(`⟨*e*⟩`*)`

Call expressions cover both calls to Go functions and to predicates. The expression in front of the parentheses determines whether the call expression is a function or predicate call. A predicate call is well-defined if the referred predicate declaration exists in scope of the predicate call, and the number and types of parameters of the call and the declaration match. Hence, if $\langle P \rangle$ is the unique name of some predicate, then the syntax of a corresponding predicate call is

$\langle e \rangle \quad ::= \quad \langle P \rangle (\langle e \rangle *)$

Figure 3.6 exemplifies how predicates are used:

```
//@ predicate sorted(nums []int) {
//@     acc(nums) && forall i, j int :: i in range nums && 0 <= j < i
//@                                     ==> nums[j] <= nums[i]
//@ }

//@ requires sorted(x)
//@ ensures forall k int :: _, k in range x ==> k >= min
func minimum(x []int) (min int) {
    return x[0]
}

//@ requires sorted(x)
//@ ensures forall k int :: _, k in range x ==> k <= max
func maximum(x []int) (max int) {
    return x[len(x) - 1]
}
```

Fig. 3.6: Examples of predicate declarations and predicate calls

A predicate is declared that asserts that a given integer slice can be accessed and it is sorted in increasing order. In the specification of the two function `minimum` and `maximum`, the predicate is called. Since this requires any input of the functions to be sorted, the minimum of a slice is always the first, and the maximum of the slice is always the last element.

3.7 Purity

Specification annotations are not allowed to change the behavior of the program, i.e. they must be side-effect free. To illustrate this requirement, we consider the following scenario:

```
func increment(x *int) int {
    *x++
    return *x
}
```

```
// This postcondition fails:
//@ ensures old(*x) == increment(x)
func decrement(x *int) int {
    *x--
    return *x
}
```

The scenario shows an increment and a decrement function for an integer pointer. The increment function is called in the postcondition of the decrement function. Thus, the integer pointer is incremented in the postcondition, hence, the specification influences the program's behavior.

We require a specification to be deterministic and free of side-effects as demonstrated above. Expressions satisfying these two properties are called `pure`. For GoRAC, we introduce the following purity definition:

Definition 11. *Let `e` be an expression. Then, `e` is considered pure if it matches one of the following cases:*

- *`e` is a constant, (composite) literal or a variable*
- *`e` is a dot expression `e1.field` and its base `e1` is pure*
- *`e` is an index expression `e1[e2]` and its base `e1` is a unary or binary expression with pure operands*
- *`e1` is a call to a pure function*

As stated in the last case, only pure function can be called in a specification. We decide to include a purity annotation in GoRAC's syntax such that users need to explicitly declare a function as pure:

$\langle d \rangle$ ::= `pure`

Purity annotations are only permitted in a function's documentation, i.e. as specification comments above a function declaration. A function annotated to be pure has to satisfy the following properties:

Definition 12. *Let `f` be a function. It satisfies a purity annotation if*

- *`f` has exactly one return parameter*
- *the body of `f` consists of only a single return statement*
- *the return statement returns a pure expression*
- *any assertion of a postcondition for `f` is a pure expression*

The built-in functions `len` and `cap` from the Go standard library are both considered pure without a corresponding annotation. This enables their use in specification and ultimately permits meaningful reasoning about properties of e.g. arrays or slices.

This concludes the chapter on the GoRAC specification language. The chapter provided a detailed description of the syntax of specification annotations for GoRAC. It is supposed to serve as a guideline when writing specification for programs that are runtime checked with GoRAC.

We would like to add a final remark that should be taken into consideration when including specification for a program. Go enforces that everything a programmer declares or imports needs to be used in the scope it was declared or imported in [14]. This has the effect that if an object is declared which is used only in specification, the program will not compile. However, we circumvent this problem with empty assignments. E.g. if a variable `x` is used only in specification, we can add the assignment "`_= x`" in the scope of the variable's declaration.

The next chapter will deal with the runtime check generation of specification annotations. It thus provides a deeper understanding of how GoRAC is constructed.

4 Runtime Check Generation

We generate runtime checks by translating specification annotations to Go code and including the code into the program that the specification reasons about. Go code that is generated for a specification annotation is called the *runtime check* for the annotation. A runtime check tests whether a condition expressed by a specification annotation holds. For GoRAC, if the condition is not satisfied, the program terminates with a respective error. We then denote the runtime check as *failed*. When a program with runtime checks runs on some input and no checks fail, we deduce that this particular program execution meets the program's specification.

Each section in this chapter details the runtime check generation for a specification construct of GoRAC. Runtime check generation is defined using *encoding functions*. An encoding of a specification constructs determines how the construct is translated to Go code. An encoding of a Go instance, e.g. a function or statement, determines how the code of a program is transformed to include runtime checks.

An encoding function thus establishes relations between sets of specification constructs and Go constructs. Recall that in Chapter 3 we detail specification clauses, assertions, and expressions as part of the specification language. In this chapter, we use $\mathcal{S}_c$, $\mathcal{S}_a$, and $\mathcal{S}_e$ to refer to the set of specification clauses, the set of specification assertions and the set of specification expressions, respectively. Moreover, we distinguish the set of Go expressions $\mathcal{G}_e$ and the set of Go statements $\mathcal{G}_s$ as defined in the Go Language Specification [14]. Now, we can introduce the following three encoding functions that define the encoding of specification clauses, specification assertions, and Go statements, respectively.

Definition 13. *Let $\mathcal{S}_c$ denote the set of specification clauses and $\mathcal{G}_s$ the set of Go statements as described in the Go Language Specification [14]. Then we can define the function*

$$\{\!|\ |\!\} : \mathcal{S}_c \to \mathcal{G}_s,\ c \mapsto s$$

that encodes a specification clause c into a Go statement s.

Definition 14. *Let $\mathcal{S}_a$ denote the set of specification assertions and $\mathcal{G}_e$ the set of Go expressions as described in the Go Language Specification [14]. Then we can define the function*

$$(\!|\ |\!) : \mathcal{S}_a \to \mathcal{G}_e,\ a \mapsto e$$

that encodes a specification assertion a into a Go expression e.

Definition 15. *Let $\mathcal{G}_s$ denote the set of Go statements as described in the Go Language Specification [14]. Then we can define the function*

$$[\![\ ]\!] : \ \mathcal{G}_s \to \mathcal{G}_s, \ s \mapsto s'$$

that encodes a Go statement s into another Go statement s'.

All three encoding functions can generate more Go members, e.g. structs or functions. We could make the generation of Go members explicit by adding more return values to the encodings. However, we decided against this in order to simplify the formalism.

We follow the structure of Chapter 3 and first address runtime check generation for specification clauses. Section 4.4 deals with quantifiers and Sections 4.5 - 4.7 concern the runtime check generation of old expressions. Permissions, predicates and purity runtime checking is explained in Sections 4.8 - 4.10.

4.0.1 Soundness

All encodings need to be defined such that if a runtime check succeeds, the program satisfies the assertion expressed by the specification. An encoding that meets this requirement is called *sound*. The following proposition states properties that are sufficient to prove soundness of an encoding:

Proposition 1. *Let s be a specification annotation, p, q and q' program states, and $panic$ the aborted program state that results from a panic. The execution of a runtime check for a specification annotation s is performed in a program state p. Furthermore, the execution of the runtime check results in program state q. If the program is run without runtime checks, the program state that is reached from p is q'. Now, we make the following proposition:*

The encodings for the runtime check of s are sound

$$\Leftrightarrow$$

$$(q \not\equiv panic \ \Rightarrow \ p \models s) \ \wedge \ (\ q' \equiv q \setminus q_{aux}\ )$$

In other words, for soundness of the encodings, it is sufficient to show that the following two requirements are fulfilled:

- *If the execution of the runtime check for s does not result in a panic, program state p satisfies s, i.e. the specified condition holds in p.*
- *If the program was executed without runtime checks, the program state q' that is reached after p is equivalent to the program state q without auxiliary state*

information. This condition can be summarized by the statement that sound runtime checks do not change a program's behavior.

In order to prove soundness of the runtime check generation, the proposition must be proven for each encoding. However, since formal proofs of soundness are beyond the scope of this thesis, we instead provide informal arguments about the correctness of an encoding.

4.1 Specification Clauses

We show the runtime check generation for each kind of specification clause in a separate subsection. Note that all code illustrations of the runtime checks state only exemplary error messages. The actual error messages implemented in GoRAC include further information on e.g. the line number of the original specification.

4.1.1 Assert Statements & Assumptions

Encoding 1. *Assert statements (or assumptions) stated with the keyword* `assert` *(or* `assume`*) are translated to Go code using the following encoding:*

$$\llbracket \texttt{assert}\ \langle a \rangle \rrbracket \rightsquigarrow \texttt{if}\ !(\!|\langle a \rangle|\!)\ \ \{\ \texttt{"ERROR"}\ \ \}$$

Assert statements (and assumptions) are checked using an if-statement. The condition of the if-statement is a negation of the encoded assertion. Thus, if the asserted assertion does not holds, the condition of the if-statement is true and the program aborts with an error message about the failed assertion. If the asserted assertion holds, the negation will be false and the program continues.

Figure 4.1 illustrates the runtime check generation for an assert statement. At the top, a program part that includes an assert statement is given. At the bottom, the code that GoRAC generates shows the runtime check for the statement. The error

```
//@ assert divisor != 0
result := 42 / divisor
```

```
if !(divisor != 0) {
    panic("Assertion violated")
}
result := 42 / divisor
```

Fig. 4.1: Runtime check for an assert statement

that occurs if the assertion does not hold is realized using `panic`. Thus, the program terminates upon failure of a runtime check.

4.1.2 Preconditions & Postconditions

Preconditions and postconditions state expectations on a function's arguments and return values. They are always part of the documentation for a function, i.e. stated in a specification comment above the function. Thus, before stating the encoding of preconditions and postconditions, we first define an encoding of functions that shows that runtime checks for preconditions and postconditions are inserted at the beginning of a function's body.

Encoding 2. *Let `name` denote an arbitrary function name, `args` the arguments of the function, `rets` the return values of the function and `s` the body of the function. Then, we encode the function as follows:*

$$
\left[\!\!\left[\begin{array}{l}
\texttt{//@ requires } \langle a \rangle \\
\texttt{//@ ensures } \langle a \rangle \\
\texttt{func name(args) (rets) \{} \\
\quad \texttt{s} \\
\texttt{\}}
\end{array}\right]\!\!\right] \quad \rightsquigarrow \quad
\begin{array}{l}
\texttt{func name(args) (rets) \{} \\
\quad \{\!\!\{ \texttt{requires } \langle a \rangle \}\!\!\} \\
\quad \{\!\!\{ \texttt{ensures } \langle a \rangle \}\!\!\} \\
\quad [\![\texttt{s}]\!] \\
\texttt{\}}
\end{array}
$$

For simplicity, we have left out potential function receivers. The encoding of methods is identical to the encoding of functions.

We encode preconditions analogously to assert statements. Since encoded preconditions are executed before any other statement of a function's body, as shown in Encoding 2, the encoding of precondition thus checks the program state at entry to the function.

Encoding 3. *Preconditions are translated to Go code using the following encoding:*

$$
\{\!\!\{ \texttt{requires } \langle a \rangle \}\!\!\} \rightsquigarrow \texttt{if !}(\!|\langle a \rangle|\!) \texttt{ \{ "ERROR" \}}
$$

For postconditions, we delay the runtime check for a postcondition until the end of a function execution using Go's `defer` statement. The actual runtime check that is deferred is the same if-statement as for an encoded assert statement. Encoding 2 states that postcondition runtime checks are inserted at the beginning of a function's body. However, the defer-statement delays the execution of the runtime check until the function terminates. It would not be adequate to insert the runtime check as the last statement of the function's body instead of using `defer`. If the function exits before reaching the end of its body, the check would not be performed. The `defer`

statement evaluates after the execution regardless of where the program flow leaves the function.

Encoding 4. *Postconditions are translated to Go code using the following encoding:*

$$\llbracket \texttt{ensures}\ \langle a \rangle \rrbracket \rightsquigarrow \begin{array}{l}\texttt{defer func() \{} \\ \quad \texttt{if !(}\langle a \rangle\texttt{) \{ "ERROR" \}} \\ \texttt{\}}\end{array}$$

Golang initializes named return parameters with zero values and returns them only on bare returns, i.e. return statements without values. If a function with named return parameters returns explicit values, the named return parameters will still be initialized with zero values (or the value they were last changed to inside the function), thus resulting in a failing postcondition. Therefore, we explicitly assigns named return parameters the returned values. Such an assignment does not change the program's behavior and ensures that the named parameters hold the values that are actually returned. The following encoding of return statements realizes the assignments:

Encoding 5. *Let* `return values` *be a return statement returning values* `values` *in a function with the named parameters* `rets`*. Then, we encode the return statement as follows:*

$$\llbracket \texttt{return values} \rrbracket \rightsquigarrow \begin{array}{l}\texttt{rets = values} \\ \texttt{return rets}\end{array}$$

In the encoding makes use of the fact that, in Go, multiple values can be assigned to multiple parameters in one lines. Thus, if the returned values are `1, "2", 3.4` and the named return parameters `i int, s string, f float32`, then we add the assignment `i, s, f = 1, "2", 3.4` when encoding a return statement.

In Figure 4.2 on the next page we find the example code discussed in Section 3.2.2 at the top and the code for runtime checking the given precondition and postcondition at the bottom. The example also demonstrates that the returned value is assigned to the named return parameter.

4.1.3 Invariants

Loop invariants express conditions that are maintained from one loop iteration to the next. In particular, a loop invariant holds upon entry to and exit from a loop. In order to ensure that the invariant holds before, during and after the loop execution, it needs to be checked at all of these program points. As invariants reason

```
//@ requires divisor != 0
//@ ensures res == x / divisor
func divide(x, divisor int) (res int) {
    return x / divisor
}
```

```
func divide(x, divisor int) (res int) {
    if !(divisor != 0) {
        panic("Precondition violated")
    }
    defer func() {
        if !(res == x / divisor) {
            panic("Postcondition violated")
        }
    }()
    res = x / divisor
    return
}
```

Fig. 4.2: Runtime checks for a precondition and a postcondition

about variables that might be declared inside the initialization statement of the loop, we define an encoding of loops that extracts the declaration of looping variables simultaneously to adding runtime checks for invariants. This establishes visibility of the variables in the scope of invariant checks before or after the loop.

Encoding 6. *Let* `init` *denote an initialization statement that is part of a for-loop, and* `cond`*,* `post` *and* `s` *the condition, post statement and body of the loop, respectively. Then, we encode the loop as follows:*

$$
\left[\!\!\left[\begin{array}{l}
\texttt{//@ invariant } \langle a \rangle \\
\texttt{for init; cond; post \{} \\
\quad\quad \texttt{s} \\
\texttt{\}}
\end{array}\right]\!\!\right]
\quad \rightsquigarrow \quad
\begin{array}{l}
\texttt{init} \\
\{\!\!\{ \texttt{invariant } \langle a \rangle \}\!\!\} \\
\texttt{for ; cond; post \{} \\
\quad\quad \{\!\!\{ \texttt{invariant } \langle a \rangle \}\!\!\} \\
\quad\quad [\![\texttt{s}]\!] \\
\texttt{\}} \\
\{\!\!\{ \texttt{invariant } \langle a \rangle \}\!\!\}
\end{array}
$$

For-loops with empty initialization or post statements are treated analogously. Range declarations are encoded similarly with the difference that instead of extracting an initialization statement, declarations of the looping variables are placed before the first invariant check.

The encoding demonstrates that encoded loop invariants are placed before and after a loop, and at the beginning of the loop body. Checking the assertion of the invariant at each of these places ensures that the loop invariant is maintained throughout the loop execution. This placement also allows us to include information about whether

the invariant failed before, during or after the loop in the panic error message of the runtime check. The disadvantage of this approach is a runtime overhead due to the two checks before and at the beginning of the loop (or after the loop if the loop is not executed) checking the same program state. Instead of the three checks, we could place one check before the loop and one at the end of the loop body. However, a check at the end of the loop body could be skipped with a continue or break statement. Consequently, we decide to use the three checks as defined in Encoding 6 despite the runtime overhead. This leaves only the encoding of invariants themselves to be defined:

Encoding 7. *Invariants are translated to Go code using the following encoding:*

$$[\![\ \textit{invariant}\ \langle a \rangle\ ]\!] \rightsquigarrow \textit{if}\ !(\!|\langle a \rangle|\!)\ \ \{\ \textit{"ERROR"}\ \ \}$$

Figure 4.3 depicts the runtime check generation for the sample program introduced in Section 3.2.3. At the top, the program with specification annotation is shown. At the bottom the runtime checking code that GoRAC generates is given. The example also shows the extraction of the initialization statement.

```
j := 9
//@ invariant i + j == 9
for i := 0; i < 10; i++  {
    j--
}
```

```
j := 9
i := 0
if !(i + j == 9) {
    panic("Invariant violated before loop execution")
}
for ; i < 10; i++  {
    if !(i + j == 9) {
        panic("Invariant violated during loop execution")
    }
    j--
}
if !(i + j == 9) {
    panic("Invariant violated after loop execution")
}
```

Fig. 4.3: Runtime check for a loop invariant

This concludes the runtime check generation for specification clauses. Note that all encodings integrate encoded assertions of the specification clauses. The encoding of these assertions is explained in the next section.

4.2 Assertions

An assertion, which is part of a specification clause, is translated to a condition for the if-statement of the runtime check of the specification clause. Before assertions are translated, GoRAC collects additional information for each of the constructs that are part of the assertion. This process is called *desugaring*. During desugaring, several actions take place:

- We save typing information of objects that are referred to in the assertion.
- We distinguish whether a call expression is a function or a predicate call.
- We transform implicit pointer dereferences such that the dereferences are explicitly stated. E.g. for a struct pointer `foo`, we can define the access to a field `bar` with an implicit dereference: `foo.bar`. When this expression is desugared, the struct pointer is explicitly dereferenced: `(*foo).bar`. Likewise, an implicit dereference of an array pointer that is used in an index expression `a[42]` is desugared into the expression with an explicit dereference `(*a)[42]`.

The information obtained by desugaring a specification assertion is used in the runtime check generation for assertions. After an assertion is desugared, its encoding is the exact translation of the specification expression into the Go language. This is defined below in Encoding 8.

Encoding 8. *Assertions that have equivalent Go expressions are encoded as follows:*

$$
\begin{array}{ll}
(\!| \langle x \rangle |\!) \rightsquigarrow x & (\!| \langle e \rangle \texttt{ <= } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ <= } (\!| \langle e \rangle |\!) \\
(\!| \langle l \rangle |\!) \rightsquigarrow l & (\!| \langle e \rangle \texttt{ > } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ > } (\!| \langle e \rangle |\!) \\
(\!| \texttt{*}\langle e \rangle |\!) \rightsquigarrow \texttt{*} (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{ >= } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ >= } (\!| \langle e \rangle |\!) \\
(\!| \texttt{+}\langle e \rangle |\!) \rightsquigarrow \texttt{+} (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{ == } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ == } (\!| \langle e \rangle |\!) \\
(\!| \texttt{-}\langle e \rangle |\!) \rightsquigarrow \texttt{-} (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{ != } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ != } (\!| \langle e \rangle |\!) \\
(\!| \texttt{!}\langle e \rangle |\!) \rightsquigarrow \texttt{!} (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{ \&\& } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ \&\& } (\!| \langle e \rangle |\!) \\
(\!| \langle e \rangle \texttt{ + } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ + } (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{ || } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ || } (\!| \langle e \rangle |\!) \\
(\!| \langle e \rangle \texttt{ - } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ - } (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{[} \langle e \rangle \texttt{]} |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{[} (\!| \langle e \rangle |\!) \texttt{]} \\
(\!| \langle e \rangle \texttt{ * } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ * } (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{.} \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{.} (\!| \langle e \rangle |\!) \\
(\!| \langle e \rangle \texttt{ / } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ / } (\!| \langle e \rangle |\!) & (\!| \langle T \rangle \texttt{\{} \langle e \rangle^* \texttt{\}} |\!) \rightsquigarrow (\!| \langle T \rangle |\!) \texttt{\{} (\!| \langle e \rangle |\!)^* \texttt{\}} \\
(\!| \langle e \rangle \texttt{ \% } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ \% } (\!| \langle e \rangle |\!) & (\!| \langle e \rangle \texttt{(} \langle e \rangle^* \texttt{)} |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{(} (\!| \langle e \rangle |\!)^* \texttt{)} \\
(\!| \langle e \rangle \texttt{ < } \langle e \rangle |\!) \rightsquigarrow (\!| \langle e \rangle |\!) \texttt{ < } (\!| \langle e \rangle |\!) &
\end{array}
$$

Now we have defined the encoding of all specification constructs for which an equivalent construct exists in the Go programming language. In addition, GoRAC includes specification constructs that are not part of Golang. Each of the next sections is dedicated to the runtime check generation of one of these additional specification constructs.

4.3 Ternary Operator

The ternary operator is used to express an if-then-else statement. Since the ternary operator as an assertion cannot be encoded as a statement but needs to be a Go expression, we encapsulate a corresponding if-then-else statement inside an anonymous function call [14].

Encoding 9. *Ternary expressions are encoded to Go code as follows:*

$$
\begin{aligned}
(\!|\langle e_1\rangle \texttt{ ? } \langle e_2\rangle \texttt{ : } \langle e_3\rangle|\!) \rightsquigarrow\ & \texttt{func() bool \{} \\
& \quad \texttt{if } (\!|\langle e_1\rangle|\!) \texttt{ \{} \\
& \quad\quad \texttt{return } (\!|\langle e_2\rangle|\!) \\
& \quad \texttt{\} else \{} \\
& \quad\quad \texttt{return } (\!|\langle e_3\rangle|\!) \\
& \quad \texttt{\}} \\
& \texttt{\}()}
\end{aligned}
$$

Figure 4.4 demonstrates the runtime check generation for ternary operators. The code at the top of the figure shows a function that computes the absolute value of a given integer. A postcondition ensures with the help of a ternary operator that

```
//@ ensures x >= 0 ? res == x : res == -x
func absoluteValue(x int) (res int)  {
    if x >= 0 {
        res = x
    } else {
        res = -x
    }
    return
}
```

```
func absoluteValue(x int) (res int)  {
    defer func() {
        if !func() {
            if x >= 0 { return res == x }
            else { return res == -x }
        }() {
            panic("Postcondition violated")
        }
    }()
    if x >= 0 {
        res = x
    } else {
        res = -x
    }
    return
}
```

Fig. 4.4: Runtime check for a ternary operator

the output of the function is the value of the input without regard to its sign. The generated runtime check as shown at the bottom of Figure 4.4 consists of a defer-statement that delays the postcondition check until function termination. Inside the defer-statement, the anonymous function call holds the encoded ternary operator that checks the assertion of the postcondition with an if-statement.

4.4 Quantifier

Similar to ternary operators, we also encode quantifiers using anonymous function calls. The anonymous function checks the condition expressed by the quantifier and returns true if the quantifier holds. We first address runtime checking of universal quantifiers and afterwards of existential quantifiers.

4.4.1 Universal Quantifier

Intuitively, a universal quantifier expresses that a given assertion holds for all quantified variables. Recall that our universal quantifier are bound by a domain. We evalute the quantified predicate on all values of the domain. When a witness is found that invalidates the predicate, false is returned. After all values are iterated, we know that no witness was found, thus the predicate holds for all values of the domain and we return true.

Encoding 10. *A universal quantifier $\langle q_u \rangle$ with quantified variables $\langle X \rangle$, domain $\langle D(X) \rangle$ and assertion $\langle a(X) \rangle$ is translated to Go code using the following encoding:*

$(\!|\langle q_u \rangle|\!) \; = \; (\!|\texttt{forall}\ \langle X \rangle \ ::\ \langle D(X) \rangle\ \texttt{==>}\ \langle a(X) \rangle|\!) \ \rightsquigarrow$

```
func() bool {
      Range(⟨D(X)⟩, ⟨X⟩)(
            if !( (|⟨a(X)⟩|) ) {
                  return false
            }
      )
      return true
}()
```

The encoding includes a call to an encoding function `Range(`$\langle D(X) \rangle$`,X)` that determines how the domain $\langle D(X) \rangle$ of the quantified variables $\langle X \rangle$ is traversed. The *range function* iterates over all values of the domain and executes for each such value the statement provided as an argument. In the encoding above, the if-statement that

composes the assertion check is input to the range function call. Let us illustrate the use of the range function by an example. The following quantifier, which states that all values of an array are positive, serves as an example.

```
//@ assert forall x int :: _, x in range arr ==> x > 0
```

The statement that checks for each quantified variable whether the quantifier condition holds is:

```
if !(x > 0) { return false }
```

This statement is input to the range function which then wraps it into a for-loop that iterates over the array such that all values of the specified domain are checked:

```
for _, x := range arr {
    if !(x > 0) {
        return false
    }
}
```

The complete runtime check for the assertion is thus:

```
if func() bool {
    for _, x := range arr {
        if !(x > 0) {
            return false
        }
    }
    return true
}() {
    panic("Assertion violated")
}
```

Recall that a quantifier domain is a formula of conjunctions and disjunctions of multiple domain constraints. The example above shows a quantifier with just a single domain constraint. We will first define an *iteration function* that handles single domain constraints. Afterwards, we proceed with the definition of a range function that can handle conjunctions and disjunction of multiple domain constraints.

For a given domain constraint, the iteration function generates code that iterates over the domain. This is realized by translating a single domain constraint in a loop declaration. The body of the loop consists of a statement that is passed into the iteration function.

Definition 16. *Let $\mathcal{G}_s$ be the set of Go statements as described in the Go Language Specification [14] and $\langle c(X) \rangle$ a constraint of a quantifier domain. Then, we can define the following iteration function:*

$$iterate(\langle c(X) \rangle) \quad : \quad \mathcal{G}_s \quad \rightarrow \quad \mathcal{G}_s$$

For a statement s, we define:

$$iterate(\langle e_1 \rangle \texttt{ < } \langle x \rangle \texttt{ < } \langle e_2 \rangle)(s) = \texttt{for } x \texttt{ := } \langle e_1 \rangle \texttt{+ 1; } x \texttt{ < } \langle e_2 \rangle \texttt{; } x\texttt{++} \{ s \}$$

$$iterate(\langle e_1 \rangle \texttt{ <= } \langle x \rangle \texttt{ < } \langle e_2 \rangle)(s) = \texttt{for } x \texttt{ := } \langle e_1 \rangle \texttt{; } x \texttt{ < } \langle e_2 \rangle \texttt{; } x\texttt{++} \{ s \}$$

$$iterate(\langle e_1 \rangle \texttt{ < } \langle x \rangle \texttt{ <= } \langle e_2 \rangle)(s) = \texttt{for } x \texttt{ := } \langle e_1 \rangle \texttt{+ 1; } x \texttt{ <= } \langle e_2 \rangle \texttt{; } x\texttt{++} \{ s \}$$

$$iterate(\langle e_1 \rangle \texttt{ <= } \langle x \rangle \texttt{ <= } \langle e_2 \rangle)(s) = \texttt{for } x \texttt{ := } \langle e_1 \rangle \texttt{; } x \texttt{ <= } \langle e_2 \rangle \texttt{; } x\texttt{++} \{ s \}$$

$$iterate(\langle x \rangle \texttt{ in range } \langle e \rangle)(s) = \texttt{for } x \texttt{ := range } \langle e \rangle \texttt{ \{ } s \texttt{ \}}$$

$$iterate(\texttt{\_, } \langle x \rangle \texttt{ in range } \langle e \rangle)(s) = \texttt{for \_, } x \texttt{ := range } \langle e \rangle \texttt{ \{ } s \texttt{ \}}$$

$$iterate(\langle x_1 \rangle \texttt{ , } \langle x_2 \rangle \texttt{ in range } \langle e \rangle)(s) = \texttt{for } x_1 \texttt{ , } x_2 \texttt{ := range } \langle e \rangle \texttt{ \{ } s \texttt{ \}}$$

The definition above describes how single terms of a conjunction or disjunction that makes up a quantifier domain are translated to Go code. Let us now look at an example of a quantifier with conjunct domains:

```
//@ assert forall x, y int :: 0 <= x < 10 && 0 <= y < 10 ==> x * y < 100
```

The quantifier states that the product of two positive integers which are smaller than ten is always less than one hundred. In order to check the quantifier assertion on the right side of the implication, both quantified variables need to be instantiated. This is realized by nesting the loops that iterate over the two domains:

```
for x := 0; x < 10; x++ {
    for y := 0; y < 10; y++ {
        if !(x * y < 100) {
            return false
        }
    }
}
```

The example implies that translating domains which are conjunctions of multiple constraints results in multiple nested loops. This nesting is performed by the range

function that was used in the encoding of universal quantifiers. In order to define the range function, we need to introduce the terms of *free* and `bound` variables. A variable is free if no domain for the variable has been declared yet. In contrast, a variable is bound if a domain has been declared for it. In other words, a variable is bound if a corresponding has been iterated by the iteration function. With the notation of free and bound variables, the range function is defined as follows:

Definition 17. *Let $\mathcal{G}_s$ be the set of Go statements as described in the Go Language Specification [14], $\langle D(X) \rangle$ a domain of a quantifier, and R a set of free quantified variables. Then we can define a range function*

$$Range(\langle D(X) \rangle, R) \quad : \quad \mathcal{G}_s \quad \rightarrow \quad \mathcal{G}_s$$

which takes a statement s as input and outputs the original statement wrapped into the range traversal of the given domain. For conjunctions we define

$$Range(\langle D_1(X) \rangle \ \&\& \ \langle D_2(X) \rangle, R)(s) \quad = \\ Range(\langle D_1(X) \rangle, R)(Range(\langle D_2(X) \rangle, R \setminus bound(D_1))(s))$$

where $bound(D_1)$ is the set of variables that are bound by any sub-domain of $D_1(X)$.

Before defining the range function case for disjunctions, we again first look at an example.

```
//@ assert forall x int :: 0 <= x <= 41 || 43 <= x <= 1337 ==> x != 42
```

The quantifier reasons about integers of disjunct intervals. When checking the quantifier assertion for the quantified integer variables, we can first test all variables bound by the first domain constraint and afterwards proceed with all variables bound by the second constraint. This results in two consecutive loops for the disjunction of the domain constraints:

```
for x := 0; x <= 41; x++ {
    if !(x != 42) {
        return false
    }
}
for x := 43; x <= 1337; x++ {
    if !(x != 42) {
        return false
    }
}
```

We can thus add the second case for the range function:

Definition 18. *Additionally to the cases of the range function defined in Definition 17, we declare the following case of the range function for disjunctions:*

$$\texttt{Range}(\langle d_1(X)\rangle \texttt{ || } \langle d_2(X)\rangle\texttt{, R)(s)} = \texttt{Range}(\langle d_1(X)\rangle\texttt{, R)(s) ; Range}(\langle d_2(X)\rangle\texttt{, R)(s)}$$

We have not yet defined how the range function declared in Definitions 17 and 18 makes use of the iteration function from Definition 16. Essentially, the range function calls the iteration function whenever a single domain constraint is encountered. However, we need to define an exceptional case when multiple domains reason about the same quantified variable, i.e. when a quantified variable is bound by multiple domains. We demonstrate this using a quantifier that imposes the restriction on an array that the first 10 elements need to be less than 100.

```
//@ assert forall x int :: 0 <= x < 10 && x in range arr ==> arr[x] < 100
```

If we translate the quantifier according to our range function and call the iteration function for each domain constraint, we obtain the following Go code:

```
for x := 0; x < 10; x++ {
    for x := range arr {
        if !(arr[x] < 100) {
            return false
        }
    }
}
```

This translation is not valid in Go since the variable x is defined twice, once in the outer and once in the inner loop. Therefore, we use a different looping variable for the inner loop and perform the check whether `arr[x] < 100` only when the inner looping variable is equal to the outer looping variable:

```
for x := 0; x < 10; x++ {
    for y := range arr {
        if y == x {
            if !(arr[x] < 100) {
                return false
            }
        }
    }
}
```

Motivated by this example, we distinguish three cases for the range function when a single domain constraint is encountered: (i) If all quantified variables of the domain constraint are free, the domain is iterated. (ii) If the domain constraint reasons about at least one variable that has already been bound, the domain is filtered for this variable as demonstrated in the example above. (iii) If all variables that the domain constraint reasons about are bound, the domain is filtered for all of

these variables. Note that a domain constraint can reason about a maximum of two quantified variables.

Definition 19. *Let $\langle C(X) \rangle$ be a domain constraint of a quantifier that bounds the quantified variables in the set X, R a set of free quantified variables, and s a statement. Then we can add the following cases to the range function defined in Definition 17:*

$$Range(\langle c(X) \rangle, R)(s) =$$

$$\begin{cases} iterate(\langle c(X) \rangle)(s) & \text{if } X \subseteq R \\ iterate(\langle c(X \setminus \{x\} \cup \{y\}) \rangle)(\texttt{if } y == x \texttt{ \{ } s \texttt{ \}}) & \text{if } X \not\subset R \wedge X \cap R \not\equiv \emptyset \\ iterate(\langle c(\{y_1, y_2\}) \rangle)(\texttt{if } y_1 == x_1 \texttt{ \&\& } y_2 == x_2 \texttt{ \{ } s \texttt{ \}}) & \text{if } X \cap R \equiv \emptyset \end{cases}$$

As mentioned in Section 3.3, boolean quantified variables can but do not need to be bounded explicitly by stating a domain. Therefore, a last case is added to the definition of the range function. In this case, the domain is empty but there are still boolean quantified variables left in the set of free variables R. The range function generates a loop that iterates over both possible truth values of the boolean quantified variable.

Definition 20. *Let b_i be a boolean variable, s a statement and _ an empty parameter. Then, we can add the following case to the range function from Definition 17:*

$$\begin{aligned} &Range(\_, \{\langle b_1 \rangle, \ldots, \langle b_n \rangle\})(s) \quad = \\ &\texttt{for } b_1 \texttt{ := range [2]bool\{true, false\} \{} \\ &\qquad Range(\_ \{\langle b_2 \rangle, \ldots, \langle b_n \rangle\})(s) \\ &\texttt{\}} \end{aligned}$$

With this being the last case of the range function definition, we conclude the encoding of universal quantifiers. Before proceeding with the encoding of existential quantifiers, we demonstrate the translation of quantifiers with boolean variables in Figure 4.5. The figure shows a quantifier with two quantified variables: one

```
//@ assert forall x int, b bool :: 42 < x < 1337 ==> (x > 0 && b)
```

```
if !func() bool {
    for x := 42 + 1; x < 1337; x++ {
        for _, b := range [2]bool{true, false} {
            if !(x > 0 && b) { return false }
        }
    }
    return true
}() { panic("Assertion ... violated.") }
```

Fig. 4.5: Runtime check for a universal quantifier with a boolean quantified variable

integer variable in between 42 and 1337, and one boolean variable. The code at the bottom depicts the code that GoRAC generates when translating the quantifier into a runtime check. Since the boolean variable is not explicitly bounded, a loop ranging over the two possible truth values is generated.

4.4.2 Existential Quantifier

An existential quantifier states that a given expression holds for at least one variable from the domain of the quantifier. As for universal quantifiers, the approach for runtime checking existential quantifiers is to iterate over all quantified variables of the domain. The encoding again uses the range function declared in Definition 17.

Encoding 11. *An existential quantifier $\langle q_e \rangle$ with quantified variables $\langle X \rangle$, domain $\langle D(X) \rangle$ and assertion $\langle a(X) \rangle$ is translated to Go code using the following encoding:*

$$(\!|\langle q_u \rangle|\!) = (\!|\texttt{exists}\ \langle X \rangle\ ::\ \langle D(X) \rangle\ \&\&\ \langle a(X) \rangle|\!) \rightsquigarrow$$

```
func() bool {
    Range(⟨D(X)⟩, X)(
        if (|⟨a(X)⟩|) {
            return true
        }
    )
    return false
}()
```

For each of the quantified variables the quantifier assertion is checked. However, contrary to universal quantifiers, the encoding of an existential quantifier returns true as soon as the expression is satisfied by a variable. In this case the quantifier holds. If this is never the case, the encoding returns false because the quantifier does not hold. Figure 4.6 exemplifies the runtime check of an existential quantifier. The

```
//@ assert len(s1) >= 10 && len(s2) >= 10 &&
//@ exists x int :: _ x in range s1 || _, x in range s2 && x == 42
```

```
if !(len(s1) >= 10 && len(s2) >= 10 && func() bool {
    for _, x := range s1 {
        if x == 42 { return true }
    }
    for _, x := range s2 {
        if x == 42 { return true }
    }
    return false
}()) { panic("Assertion violated.") }
```

Fig. 4.6: Runtime check for an existential quantifier

given assertion checks whether there is an element that is equal to 42 in any of two given slices that each need to hold at least 10 integers. The figure further shows that quantifiers can be embedded into longer assertions.

4.4.3 Optimizations

The encoding of a quantifier with a conjunction as its domain can result in deeply nested loops. Such nested loops lead to expensive run times. Therefore, it is desirable to optimize the generated code of the runtime checks for quantifiers. This subsection motivate and illustrate several optimizations that are realized in GoRAC.

Let us consider the following code that might be produced when generating a runtime check for some quantifier.

```
for x := 0; x < n; x++ {
    for y := 0; y < m; y++ {
        if y == x {
            if x < 0 {
                return true
            }
        }
    }
}
```

The nested loops have a runtime of $\mathcal{O}(\texttt{n} \times \texttt{m})$. However, for every execution of the outer loop, the if-condition `y == x` only holds at most n times. Thus, the inner check whether `x < 0` is executed at most n times. This means that there are $\texttt{n} \times (\texttt{m} - 1)$ superfluous executions of the inner loop.

Since this is not efficient, we aim to optimize certain nested loops. In the example above, we replace the inner loop together with the filtering condition `y == x` by a simple check whether `x` is in the range defined by the loop header. The optimized code given below runs in $\mathcal{O}(n)$ which is a significant improvement to the non-optimized version.

```
for x := 0; x < n; x++ {
    if x >= 0 && x < m {
        if x < 0 {
            return true
        }
    }
}
```

We define an optimization function in order to transform runtime checks of quantifiers into their optimized versions:

Definition 21. *Let $\mathcal{G}_s$ denote the set of Go statements as described in the Go Language Specification [14]. Then the function*

$$\lfloor\ \rfloor : \mathcal{G}_s \to \mathcal{G}_s,\ s \mapsto \lfloor s' \rfloor$$

translates the Go statement s into an optimized statement s' with respect to a faster runtime.

In the following, we give a list of optimizations. Each optimization is defined for a loop that contains a filtering condition. The placeholder `s` denotes an arbitrary statement that makes up the body of the if-condition. The left side shows the original code, the right side the corresponding optimized code.

Optimization 1. *Let `a` be an array or slice, and `x` an integer variable. Then filtering index `i` for the value of variable `x` is optimized as follows:*

```
for i := range a {                    ⇝    if x >= 0 && x < len(a) {
    if i == x {                                s
        s                                  }
    }
}
```

Optimization 2. *Let `a` be an array or slice, and `x` and `y` integer variables. Then filtering index `i` and value `v` for the values of variables `x` and `y` is optimized as follows:*

```
for i, v := range a {                 ⇝    if x >= 0 && x < len(a)
    if i == x && v == y {                      && a[x] == y {
        s                                      s
    }                                      }
}
```

Optimization 3. *Let `m` be a map and `x` a variable. Then filtering key `k` for the value of variable `x` is optimized as follows:*

```
for k := range m {                    ⇝    if _, ok := m[x]; ok {
    if k == x {                                s
        s                                  }
    }
}
```

Optimization 4. *Let* `m` *be a map and* `x` *and* `y` *variables. Then filtering key* `k` *and value* `v` *for the values of variables* `x` *and* `y` *is optimized as follows:*

```
for k, v := range m {
    if k == x && v == y {
        s
    }
}
```

⇝

```
if v, ok := m[x]; ok
    && v == y {
    s
}
```

Optimization 5. *Let* `x` *be a variable. Then filtering a numeric loop with loop variable* `i` *for the value of variable* `x` *is optimized as follows:*

```
for i := e1; i ∘2 e2; i++ {
    if i == x {
        s
    }
}
```

⇝

```
if e1 ∘1 x && x ∘2 e2 {
    s
}
```

where the domain constraint is $e_1 \circ_1 i \circ_2 e_2$ *with* $\circ_1, \circ_2 \in \{<, <=\}$.

When a runtime check is generated for a domain constraint, it is first checked whether an optimization can be applied. If so, the optimized code is returned. Note that filtering for values in data structures without specified indexes or keys cannot be optimized. This is due to the fact that we would need to guess the key that belongs to the filtered value in order to shorten the iteration over all values in the data structure, which is not possible.

In theory, we can think of even more quantifier optimizations. For instance, all numeric ranges for the same quantified variable can be summarized into one:

```
for i = a; i < b; i++ { for i = c; i < d; i++ { s }}
```

is optimized into

```
for i = math.Max(a, c); i < math.Min(b, d); i++ { s }
```

In practice, we implemented only optimizations that we assumed to happen frequently in order to reduce the complexity of the generated code.

The presented strategy for runtime check generation of quantifiers has the benefit that every element that satisfies the given constraints, e.g. every element of a data structure or in a numeric range, is tested against the quantifier's assertion. This is a very safe strategy for checking quantifiers that does not leave room for doubt whether the quantifier holds as long as the implementation succeeds. On the other hand,

the strategy also has the weakness that for very large ranges it is not so efficient. Under the assumption that the original program declares only meaningful data structures, iterating all of their values will always be rational. However, quantifier with integer bounds might result in very large ranges to be iterated that lead to a significant decrease in efficiency when executing the program with runtime assertion checks in comparison to the original program. In such a case, a different strategy for checking quantifiers could be sampling of numeric values to check the quantifier's assertion. We have not implemented a sampling strategy but still wanted to address this problem as part of the thesis and provide a solution idea for it.

4.5 Old Expressions

Since there are different semantics for shared and exclusive old variables, the runtime check generation for old expressions differs depending on whether the expression contains shared or exclusive old variables. In the following, we use the word *exclusive old expression* if we refer to an expression used in old that contains an exclusive variable. E.g. if `foo` is an exclusive struct variable, then `foo.bar` is an exclusive expression. Similarly, the word `shared old expression` is used for expressions that contain a shared variable. Any complex expression can be divided into subexpressions that are either shared or exclusive. We will first address the runtime checking of shared old expressions and afterwards deal with runtime check generation for exclusive old expressions. Since runtime checking of exclusive old expressions is more complex than generating runtime checks for shared old expressions, we have dedicated the whole next section to exclusive old expressions while runtime checking shared old expressions is described in the next subsection.

4.5.1 Shared Variables in Old Expressions

Intuitively, we convert a shared old expression into a Go expression by replacing it with a variable that stores the old value of the expression. Consider the Go program in Figure 4.7: Two integers `x` and `y` given. A pointer `p` is declared which points to

```
x, y := 0, 1
p := &x //@ shared: p
L:
p = &y
x, y := 2, 3
P:
//@ assert old[L](*p) == 0
```

Fig. 4.7: Example of a shared variable used in an old expression

x at label L and afterwards to y at label P. In between the two labels, the values of the two integers also change. An assert statement after program point P uses an old expression. The old expression reasons about the old value of *p at label L. As the variable p is shared, the old expression old[L](*p) is evaluated in the heap at program point L. At this program point, the pointer holds the address of x whose value is zero. Hence, the assert statement should succeed. For checking the assertion at runtime, we make use of a helper variable that saves the old value of *p and is later used instead of the shared old expression in the encoded assert statement. The following figure shows the code that GoRAC generates for the example in Figure 4.7:

```
var oldP int // helper variable
x, y := 0, 1
p := &x
L:
oldP = *p // assign old value
p = &y
P:
if !(oldP == 0)) { panic(" ... ") } // uses helper variable
```

Fig. 4.8: Runtime check for the old expression with a shared variable of Figure 4.7

The helper variable is called oldP and has the same type as the operand of the old expression. At label L, to which the old expression refers, the value of the shared old expression is assigned to the helper variable. Finally, the helper variable replaces the shared old expression in the assertion.

The example illustrates how we generate runtime checks for shared old expressions. Now, we describe a general approach with the help of a function that maps any shared old expression to a distinct variable:

Definition 22. *Let $\mathcal{S}_e$ be the set of specification expressions, L the set of labels and $\mathbb{X}$ the set of Go variables. Then, we can define the following function*

$$oldVariable : \mathcal{S}_e \times L \to \mathbb{X}, \quad (e, L) \mapsto x_{(e, L)}$$

returning a variable corresponding to expression e at label L.

The variable $x_{(e, L)}$ saves the value of the specification expression e at label L. We call this variable the *old variable* for old expression e. After declaring the old variable, the encoding of shared old expressions is simply a replacement of the old expression by the corresponding old variable.

Encoding 12. *Shared old expressions are translated to Go code using the following encoding:*

$$(\!| old[L](\langle e \rangle) |\!) \rightsquigarrow oldVariable(e, L) = x_{(e, L)}$$

In order to include the declaration of an old variable at the beginning of a function, we need to define an encoding of functions additionally to Encoding 2 that handles functions with preconditions and postconditions. The declaration of an old variable that has type $\texttt{T}_\texttt{e}$ is included at the beginning of a function whose specification uses the shared old expression `old[L](e)`. At the label `L`, the value of `e` is assigned to the variable $\texttt{x}_{(\texttt{e},\ \texttt{L})}$. Note that for unlabeled old expressions, this assignment is performed directly after the variable declaration at the beginning of the function.

Encoding 13. *Let `name` denote an arbitrary function name, `args` the arguments of the function, `rets` the return values of the function, and s_1 and s_2 lists of statements. Let further `old[L](e)` be an old expression that is used in the specification for the function, where `L` denotes a label, `e` a shared old expression, and we have $(\!|\texttt{old[L]}(\langle e \rangle)\ |\!) = x_{(e,\ L)}$. Then, we encode the function as follows:*

$$
\begin{array}{|l|l|l|}
\hline
\left[\!\!\left[\begin{array}{l}
\texttt{func name(args) (rets) \{} \\
\quad s_1 \\
\quad \texttt{L:} \\
\quad s_2 \\
\texttt{\}}
\end{array}\right]\!\!\right] & \rightsquigarrow & \begin{array}{l}
\texttt{func name(args) (rets) \{} \\
\quad \texttt{var } x_{(e,\ L)}\ T_e \\
\quad [\![s_1]\!] \\
\quad \texttt{L:} \\
\quad x_{(e,\ L)} = \{\!|\langle e \rangle|\!\} \\
\quad [\![s_2]\!] \\
\texttt{\}}
\end{array} \\
\hline
\end{array}
$$

For simplicity, we have left out potential function receivers. The encoding of methods with old expressions in their specification is identical to the encoding of functions.

Hereby, we have finished the runtime checking for shared old expressions. The runtime check generation of shared old expressions has been fully implemented in GoRAC . The tool supports the use of shared old expressions in post-conditions, assert statements, assumptions, and invariants as mentioned in Section 3.4.3 with only a minor restriction that is further discussed in Section 4.7.2.

4.6 Exclusive Variables in Old Expressions

Generating runtime checks for exclusive old expressions requires a more elaborate approach than the generation for shared old expressions. As for shared old expressions, we make use of the preliminary definitions given in Section **??**. We will start with an example to point out the difficulties of checking exclusive old expressions. The example is translated to Go code in order to illustrate the idea of the runtime check generation. Afterwards, multiple sections will build up the general approach. In the end, Section 4.6.6 presents the final encoding.

```
x, y := 0, 1
p := &x //@ exclusive: p
L:
p = &y
x, y := 2, 3
P:
//@ assert old[L](*p) == 1
```

Fig. 4.9: Program with a heap-dependent exclusive old expression

Recall the semantics of exclusive variables and the definition of heap-independent and heap-dependent as introduced in Section 3.4.1: Exclusive variables in old expressions always evaluate to their current value. For any heap-independent expression used in an exclusive old expression, this means that we can simply replace the exclusive old expression by the expression itself. The expression will thus be evaluated at the point of the old expression and result in its current value. This section describes the encoding of exclusive old expressions with heap-dependent expressions. We will start with an example to demonstrate the challenges we face when runtime checking heap-dependent exclusive old expressions.

Figure 4.9 illustrates a similar scenario as the example of shared old expressions given in Figure 4.7. The difference between the two programs is that the pointer variable `p` is now exclusive. This also results in a different assertion at the end of the program because of the different semantics for shared and exclusive variables. The old expression used in the assertion at label `P` reasons about the old value of `*p` at label `L`. Since `p` is exclusive, `p` evaluates to its current value which is the address of `y` at program point `P`. Dereferencing this address at label `L` yields `1`.

For the runtime check generation of the assertion given above, a natural assumption is to proceed similarly to heap-independent expressions and simply evaluate `e` at program point `P`. However, Figure 4.9 shows that this is not possible since evaluating `*p` at label `P` reflects the change that has been made to `y` in between `L` and `P`. Hence, the expression `*p` is equal to `3` at label `P` instead of `1` as required by the exclusive old semantics.

To correctly reflect the desired semantics, a heap lookup at program point `L` for the pointer stored in `p` has to be performed. However, for this lookup, the pointer value at program point `P` should be used which is not known yet at program point `L`. In other words, at program point `L` the value that `p` will have at program point `P` has to be predicted.

The example thus shows that for an exclusive variable, we need to already predict in the old state which values can be assigned to the variable in the future. We model

this prediction using maps which save all potential future values of variables. I.e. we do not precisely determine the future value of a variable but over-approximate the problem and store all values that could potentially be assigned to it. The maps are used to lookup the value an old expression had at an earlier program point. The keys of the lookup maps are the addresses of the values which are saved. An address uniquely identifies variables over all program points even if the variable's value changes. Therefore, we can lookup a value that a variable had at a previous program point using the address of the variable.

Let us outline the approach using the example given in Figure 4.9. We save all addresses of values that can potentially be assigned to the old expression in a set called `candidates`. Since the operand of the expression `*p` is of type integer, all integer variables are potential candidates. Thus, we save the addresses of `x` and `y` in the set `candidates`. At label `L`, the current values of all candidates from this set are saved in a map called `lookup`. The keys of this `lookup` map are the candidate addresses `&x` and `&y`. They are mapped to the values that `x` and `y` have at program point `L`, respectively. For the runtime check of the assertion, we replace the old expression by a map lookup using `p` as the key. Since the evaluation of `p` at program point `P` yields the address of `y`, we lookup the value belonging to the address of `y` in the `lookup` map. The lookup returns 1 which is the value of `y` at label `L`. The generated code for this example is given in Figure 4.10. The `candidate` set is modeled as a map of addresses to boolean values which are always `true` because Go does not have built-in sets.

We can proceed with the description of a general approach of runtime checking an exclusive old expression e_s. The approach can be divided into four steps:

- **Candidate map declaration**: We determine all types of subexpressions of e_s and declare a candidate map for each of these types. We refer to these types as *candidate types*. A value that e_s could evaluate to is called a *candidate value*.

```
candidates := map[*int]bool{}
lookup := map[*int]int{}
x, y := 0, 1
candidates[&x] = true
candidates[&y] = true
p := &x
L:
for c := range candidates {
    lookup[c] = *c
}
p = &y
x, y := 2, 3
P:
if !(lookup[p] == 1) { panic(" ... ") }
```

Fig. 4.10: Program from Figure 4.9 with runtime checks

- **Retrieving candidates**: We inspect all Go expressions that are part of the function in which the old expression $\texttt{e}_s$ occurs and decide which ones hold candidate values for $\texttt{e}_s$. These expressions are called *candidates* and are saved in the respective candidate map.
- **Lookup map declaration**: We declare a lookup map for $\texttt{e}_s$ and fill it with the values that the respective candidates have at the label $\texttt{e}_s$ refers to.
- **Performing lookups**: The exclusive old expression $\texttt{e}_s$ is replaced by a lookup in the appropriate lookup map.

Note that if $\texttt{e}_s$ contains a nested old expression, we need to apply the four steps above to the nested old expression as well. The next subsections each deal with one of the four steps outlined above: Section 4.6.1 describes how to decide which types are candidate types for a given old expression. Subsection 4.6.2 explains how candidates for an old expression are retrieved. Subsection 4.6.3 shows the declaration of lookup maps, and Subsection 4.6.4 defines a lookup function for the exclusive old expression. The last part, Subsection 4.6.5, completes the general procedure by combining the presented functions.

4.6.1 Candidate Types

We define a function `candidateTypes` that returns all subexpressions with their respective type for a given old expression. The function `candidateTypes` takes the old expression $\texttt{e}_s$ as input and returns a set of triples that each consist of a subexpression $\texttt{e}_i$ of $\texttt{e}_s$, a type $\texttt{T}_i$, and a label `L`. The type $\texttt{T}_i$ is the (candidate) type for subexpression $\texttt{e}_i$. In other words, all candidate values for the expression $\texttt{e}_i$ at the label `L` will be of type $\texttt{T}_i$. The label `L` corresponds to the label of the old expression the subexpression is a part of. Since nested old expressions with different labels are allowed, not all subexpressions need to have the same corresponding label.

Definition 23. *Let $\mathcal{S}_e$ be the set of specification expressions, L the set of labels and $\mathbb{T}$ the set of Go types. Then, we can define the following function*

$$\mathit{candidateTypes} \;:\; \mathcal{S}_e \times L \;\to\; \mathbb{T} \times \mathcal{S}_e \times L$$

$$(e_s,\ L) \;\mapsto\; \{(T_1,\ e_{s1}',\ L_1),\ \ldots,\ (T_n,\ e_{sn}',\ L_n)\}$$

that maps a given tuple of specification expression and label to a set of triples each containing a candidate type, subexpression, and label. The definition of the function is split into multiple cases:

$$candidateTypes(e_s,\ L) =$$

$$\begin{cases} \{\} & \text{if } e_s \in \mathbb{X} \times \mathbb{L} \\ \{\ (T_{e_s},\ e_s,\ L)\ \} \cup candidateTypes(e_s\text{'},\ L) & \\ & \text{if } e_s \equiv *e_s\text{'} \\ candidateTypes(e_s\text{'},\ L) & \\ & \text{if } e_s \equiv e_s\text{'}.f \\ candidateTypes(e_s\text{'},\ L\text{'}) & \\ & \text{if } e_s \equiv old[L\text{'}](e_s\text{'}) \\ candidateTypes(e_s\text{'},\ L) \cup candidateTypes(e_s\text{"},\ L) & \\ & \text{if } e_s \equiv e_s\text{'}[e_s\text{"}],\ e_s\text{' } array \\ \{\ (T_{e_s},\ e_s,\ L)\ \} \cup candidateTypes(e_s\text{'},\ L) \cup candidateTypes(e_s\text{"},\ L) & \\ & \text{if } e_s \equiv e_s\text{'}[e_s\text{"}],\ e_s\text{' } slice \\ \{\ (T_{e\text{'}_s},\ e_s,\ L)\ \} \cup candidateTypes(e_s\text{'},\ L) \cup candidateTypes(e_s\text{"},\ L) & \\ & \text{if } e_s \equiv e_s\text{'}[e_s\text{"}],\ e_s\text{' } map \end{cases}$$

The definition of the candidate types function exploits the fact that a constant, literal or variable used in an exclusive old expression has to evaluate to its constant and current value, respectively. Hence, for constants and variables we do not add any candidate types. For a dereference `*e`$_s$`'`, a candidate type is the type of the value to which the dereferenced pointer refers, i.e. the type of `e`$_s$. This type `T`$_{e_s}$ is returned in a triple with the current specification expression, to denote the origin of this candidate type, and the current specification label. Since we might have to save further candidate types for nested exclusive expressions, we afterwards recurse the expression that is dereferenced. For dot expressions `e`$_s$`'.f`, we recursively call `candidateTypes` on the structure `e`$_s$`'`. As mentioned in Section 4.2, desugaring removes implicit dereferences. Hence, the recursive call will either return no candidate types if the structure is a variable that evaluates to its current value, or return a candidate type for the dereference. If we encounter a nested old expression `old[L'](e`$_s$`')`, the label is updated in the next recursive step to the label `L'` of the inner old expression. For index expressions `e`$_s$`'[e`$_s$`"]`, we need to differentiate whether an array, slice or map is indexed: Since arrays are values in Go and therefore not treated as pointers, we simply call the `candidateTypes` function on the array `e`$_s$`'` and the index of the expression `e`$_s$`"`. In contrast, slices are treated as pointers. Thus, for slices, we return a candidate type that is the type of the index expression and also recurse the slice and the index. Maps are treated likewise, except that the candidate type is the map type such that information on the types of both the keys and the values of the map are returned. In the next section, candidate types are used to retrieve candidates. In Section 4.6.3, candidate types are needed to declare correct lookup maps.

4.6.2 Candidates

In the next step of the runtime check generation for exclusive old expressions, we collect addresses of heap locations that are potentially referred to in an exclusive old expression. Recall that we call such a heap location a candidate for an old expression. A candidate is an expression that evaluates to a candidate value, i.e. a value that the exclusive old expression could evaluate to. Thus, all candidates are of a candidate type that was described in the previous subsection. In order to determine all candidates for an old expression $\texttt{e}_s$, we call a function named `candidates` on all Go statements of a function before the specification annotation in which $\texttt{e}_s$ occurs.

Definition 24. *Let $\mathbb{T}$ be the set of Go types, $\mathcal{G}_e$ the set of Go expressions and A the set of addresses. Then, we can define the following function*

$$\mathit{candidates} \;:\; \mathbb{T} \times \mathcal{G}_e \;\rightarrow\; A^k, \quad (t,\ e_g) \mapsto \{a_1,\ \ldots,\ a_k\}$$

which determines the set of addresses of all candidates for a single Go expression e_s and candidate type t. An address from this set is denoted as a_i.

For every statement, the `candidates` function is called with each unique candidate type that is part of any triple returned by `candidateTypes`. The function `candidates` returns for a given Go statement or expression $\texttt{e}_g$ and type `t` all addresses of *addressable* subexpressions of $\texttt{e}_g$ that are of type `t`. The Go Language Specification defines an expression as addressable if it is "either a variable, pointer indirection, or slice indexing operation; or a field selector of an addressable struct operand; or an array indexing operation of an addressable array" [14]. Note that a call of the `candidates` function on a single Go expression can return several addresses: Let `t` be an integer type and $\texttt{e}_g$ a struct pointer with two integer fields, then the addresses of both fields are returned. We consider only addressable subexpressions since candidate values of heap-dependent old expressions can either be assigned or constructed from addressable expressions. E.g. for an expression `5 + *a`, the `candidates` function would not return the address of the entire expression but only the address of `a`. After having called the `candidates` function with type `t` on all Go statements before the specification annotation in which $\texttt{e}_s$ occurs, we receive the addresses $\{\texttt{a}_1, \ldots, \texttt{a}_k\}$ of all candidate values for $\texttt{e}_s$.

In the following, we adopt the naming convention for statements and expressions of the Go Language Specification [14]. Any statement or expression which is not handled by a case stated below results in the `candidates` function returning the empty set. Because Go consists of a variety of different kinds of statements or expressions, we split the definition of `candidates` into several parts: Subsection 4.6.2 starts by determining candidates from expressions which are operands. Then, Subsection 4.6.2 addresses the candidate retrieval of primary expressions. Subsection 4.6.2 concludes the definition of the `candidates` function by dealing with Go statements.

Operand Candidates

We first define the `candidates` function for operands, the most basic building blocks of expressions. An operand can be a literal, an identifier denoting a constant, variable or function, or a parenthesized expression. A qualified identifier is an identifier with a package name prefix[14]. Note that we do not consider basic or function literals since they are not addressable.

⟨Operand⟩ ::= Literal | OperandName | (Expression)

⟨Literal⟩ ::= CompositeLit

⟨OperandName⟩ ::= Identifier | QualifiedIdentifier

When calling the `candidates` function on an `Identifier` or a `QuantifiedIdentifier`, we check whether it's type is the candidate type. If so, the function returns an address to the identifier. For parenthesized expressions we simply call `candidates` on the enclosed expression:

$$\texttt{candidates(t, } e_g\texttt{)} = \begin{cases} \{\&e_g\} & \texttt{if } e_g \texttt{ (Qualified) Identifier} \wedge T_{e_g} \equiv \texttt{t} \\ & \wedge\ T_{e_g} \texttt{ not struct, slice, array or map type} \\ \texttt{candidates(t, } e_g\texttt{')} & \texttt{if } e_g \equiv \texttt{(} e_g\texttt{')} \end{cases}$$

Next, we describe the `candidates` function for composite literals and identifiers referring to variables of type struct, array, slice or map: For an expression e_g of type struct, we write e_g`.Fields` to refer to all fields of e_g. This notion applies to both composite struct literals and struct identifiers. Since any field of a struct might be a candidate value for an exclusive old expressions, the `candidates` function is recursively called on all the struct fields when a composite struct literal or struct identifier is encountered. In the first case, the struct identifier or literal itself is a candidate and thus its address is saved before recursing its fields. The second case handles struct identifiers or literals which are not candidates themselves but might contain fields that are candidates:

$$\texttt{candidates(t, } e_g\texttt{)} = \begin{cases} \{\&e_g\} \cup \bigcup_{\texttt{f} \in e_g\texttt{.Fields}} \texttt{candidates(t, f)} & \texttt{if } T_{e_g} \equiv \texttt{t} \wedge e_g \texttt{ Identifier or} \\ & \texttt{CompositeLit} \wedge T_{e_g} \texttt{ struct type} \\ \bigcup_{\texttt{f} \in e_g\texttt{.Fields}} \texttt{candidates(t, f)} & \texttt{if } T_{e_g} \not\equiv \texttt{t} \wedge e_g \texttt{ Identifier or} \\ & \texttt{CompositeLit} \wedge T_{e_g} \texttt{ struct type} \end{cases}$$

For identifiers or composite literals of type array or slice, we differentiate three cases: The first case handles candidates of type array. If an array identifier or composite literal is encountered, the address of this candidate array is saved. The second case handles candidates of type slice. A slice is a dynamically-sized view at a particular offset into an underlying array. When determining candidates of type slice, the offset and size of the slice are flexible, only the type of the underlying array is known. For a slice (or array) type `t`, we refer to the type of slice (or array) members with `t.memberType`. Now, if an array of type $\texttt{T}_{\texttt{e}_g}$ is encountered whose members are of the same type as the slice members, i.e. $\texttt{T}_{\texttt{e}_g}\texttt{.memberType} \equiv \texttt{t.memberType}$, any sub-section of this array is a potential candidate due to the flexible sizes and offsets of slices. Saving all sub-sections for an array of length n would result in $\mathcal{O}(n \times \frac{n-1}{2}) = \mathcal{O}(n^2)$ candidates. Therefore, the `candidates` function is defined to save every entry of the array instead. This results in only $\mathcal{O}(n)$ candidates. The lookup of a slice will later determine the starting position and length of the slice. The same procedure is applied if we are looking for candidate slices and encounter a slice. The third case handles candidates which have neither array nor slice types. When encountering an array or slice, every member is inspected for potential candidates:

$$
\texttt{candidates(t, e}_g\texttt{)} = \begin{cases}
\{\&\texttt{e}_g\} & \texttt{if } \texttt{T}_{\texttt{e}_g} \equiv \texttt{t} \wedge \texttt{e}_g \texttt{ Identifier or} \\
 & \texttt{CompositeLit} \wedge \texttt{T}_{\texttt{e}_g} \texttt{ array type} \\
\bigcup_{\texttt{i} \in \texttt{0} \ldots \texttt{len(e}_g\texttt{)}} \{\&\texttt{e}_g\texttt{[i]}\} & \texttt{if e}_g \texttt{ Identifier or CompositeLit} \\
 & \wedge \texttt{T}_{\texttt{e}_g} \texttt{ array or slice type} \wedge \texttt{t slice type} \\
 & \wedge \texttt{T}_{\texttt{e}_g}\texttt{.memberType} \equiv \texttt{t.memberType} \\
\bigcup_{\texttt{i} \in \texttt{0} \ldots \texttt{len(e}_g\texttt{)}} & \texttt{if e}_g \texttt{ Identifier or CompositeLit} \\
\texttt{candidates(t, e}_g\texttt{[i])} & \wedge \texttt{T}_{\texttt{e}_g} \texttt{ array or slice type} \\
 & \wedge \texttt{t } \textbf{\textit{not}} \texttt{ array or slice type}
\end{cases}
$$

When looking for candidates of a map type, we refer to the key type of the candidate map type as *candidate key type* and to the value type of the candidate map type as *candidate value type*. Handling map identifiers or composite map literals results in two cases of the `candidates` function. The first case handles a map whose types of keys and values match the candidate key type and candidate value type, respectively. In order to save all the map entries as candidates, we simply save the map's address. As later subsections will show, candidates of map type will be treated in a way that all their entries will be considered. The second case shows that if a map is inspected during a search for candidates of some type other than a map type, each value in the map is recursively inspected. There is no case that inspects keys whose types

match the candidate type `t` since keys cannot be candidates (neither can indices of arrays or slices):

$$\texttt{candidates(t, } e_g\texttt{)} = \begin{cases} \{\&e_g\} & \texttt{if } e_g \texttt{ Identifier or CompositeLit} \\ & \wedge\, T_{e_g} \texttt{ map type} \\ & \wedge\, \texttt{t map type} \\ & \wedge\, T_{e_g}\texttt{.keyType} \equiv \texttt{t.keyType} \\ & \wedge\, T_{e_g}\texttt{.valueType} \equiv \texttt{t.valueType} \\ \bigcup_{k \in e_g\texttt{.Keys}} \texttt{candidates(t, } e_g\texttt{[k])} & \texttt{if } e_g \texttt{ Identifier or CompositeLit} \\ & \wedge\, T_{e_g} \texttt{ map type} \\ & \wedge\, \texttt{t } \textbf{\textit{not}} \texttt{ map type} \end{cases}$$

Primary Expression Candidates

As a single operand is considered to be a primary expression. Primary expressions are the most basic expressions in Go. The primary expressions we consider for the definition of `candidates` are operands, index-, slice-, dot-, and call expressions:

⟨PrimaryExpr⟩ ::= Operand | IndexExpr | SliceExpr | DotExpr | CallExpr

⟨IndexExpr⟩ ::= PrimaryExpr [Expression]

⟨SliceExpr⟩ ::= PrimaryExpr [:]

⟨DotExpr⟩ ::= PrimaryExpr . Expression

⟨CallExpr⟩ ::= PrimaryExpr (Expression*)

Given an array `a`, a valid index expression is `a[42]` and an example of a slice expression is `a[:]`. For a struct `s`, a dot expression `s.f` denotes an access to its field `f`. Examples of call expressions are function calls such as `foo()` or method calls such as `foo.bar()`.

Index-, slice- and dot expressions are handled by the following part of the `candidates` definition. For index and dot expression, we first check whether the expression has the candidate type. If so and the expression is addressable, the address of the expression is saved. In any case, the `candidates` function is called on the underlying data structure. The index of an index expression is not recursed since an array or slice index cannot be a candidate, analogously to a map key. We have seen how slices (respectively expressions of type slice) are handled above, now we look at expressions that create a new slice from an existing array or slice. For these slice expressions, we simply recurse on the data structure used to declare the slice to search for candidates:

$$\texttt{candidates(t, e}_g\texttt{)} = \begin{cases} \&\texttt{e}_g \cup \texttt{candidates(t, e}_g\texttt{'}\texttt{)} & \text{if } \texttt{e}_g \equiv \texttt{e}_g\texttt{'[e}_g\texttt{"] IndexExpr} \wedge \texttt{T}_{\texttt{e}_g} \equiv \texttt{t} \\ \texttt{candidates(t, e}_g\texttt{'}\texttt{)} & \text{if } \texttt{e}_g \equiv \texttt{e}_g\texttt{'[e}_g\texttt{"] IndexExpr} \wedge \texttt{T}_{\texttt{e}_g} \not\equiv \texttt{t} \\ \&\texttt{e}_g \cup \texttt{candidates(t, e}_g\texttt{'}\texttt{)} & \text{if } \texttt{e}_g \equiv \texttt{e}_g\texttt{'.field DotExpr} \wedge \texttt{T}_{\texttt{e}_g} \equiv \texttt{t} \\ \texttt{candidates(t, e}_g\texttt{'}\texttt{)} & \text{if } \texttt{e}_g \equiv \texttt{e}_g\texttt{'.field DotExpr} \wedge \texttt{T}_{\texttt{e}_g} \not\equiv \texttt{t} \\ \texttt{candidates(t, e}_g\texttt{'}\texttt{)} & \text{if } \texttt{e}_g \equiv \texttt{e}_g\texttt{'[:] SliceExpr} \end{cases}$$

For a call expression $\texttt{e}_g$ we denote its input parameters with $\texttt{e}_g\texttt{.InputParams}$. For a method call, we further use the notation $\texttt{e}_g\texttt{.Receiver}$ to refer to the data structure the method was called on. We know that the call expression is a method call if the receiver is not empty. Since both function and method calls might change data associated with parameters or the receiver data structure, we need to reinspect the parameters and receivers for candidates after a call has returned. This results in two cases for both function and method calls: One, in which the return type of the call has the candidate type and is therefore saved in addition to recursing the parameters (and receivers). And another one, in which we only execute the `candidates` function on the parameters (and receivers) since the result of the call expression is not a candidate:

$$\texttt{candidates(t, e}_g\texttt{)} = \begin{cases} \{\&\texttt{e}_g\} \cup \bigcup_{\texttt{p} \in \texttt{e}_g\texttt{.InputParams}} \texttt{candidates(t, p)} & \text{if } \texttt{e}_g \texttt{ CallExpr} \\ & \quad \wedge \texttt{T}_{\texttt{e}_g} \equiv \texttt{t} \wedge \texttt{e}_g\texttt{.Receiver} \equiv \texttt{nil} \\ \bigcup_{\texttt{p} \in \texttt{e}_g\texttt{.InputParams}} \texttt{candidates(t, p)} & \text{if } \texttt{e}_g \texttt{ CallExpr} \\ & \quad \wedge \texttt{T}_{\texttt{e}_g} \not\equiv \texttt{t} \wedge \texttt{e}_g\texttt{.Receiver} \equiv \texttt{nil} \\ \{\&\texttt{e}_g\} \cup \texttt{candidates(t, e}_g\texttt{.Receiver)} \cup & \\ \quad \bigcup_{\texttt{p} \in \texttt{e}_g\texttt{.InputParams}} \texttt{candidates(t, p)} & \text{if } \texttt{e}_g \texttt{ CallExpr} \\ & \quad \wedge \texttt{T}_{\texttt{e}_g} \equiv \texttt{t} \wedge \texttt{e}_g\texttt{.Receiver} \not\equiv \texttt{nil} \\ \texttt{candidates(t, e}_g\texttt{.Receiver)} \cup & \\ \quad \bigcup_{\texttt{p} \in \texttt{e}_g\texttt{.InputParams}} \texttt{candidates(t, p)} & \text{if } \texttt{e}_g \texttt{ CallExpr} \\ & \quad \wedge \texttt{T}_{\texttt{e}_g} \not\equiv \texttt{t} \wedge \texttt{e}_g\texttt{.Receiver} \not\equiv \texttt{nil} \end{cases}$$

Primary expressions can be combined into more complex expressions using unary and binary operators:

⟨Expression⟩ ::= PrimaryExpr | UnaryExpr | BinaryExpr

⟨UnaryExpr⟩ ::= unary_op UnaryExpr

⟨BinaryExpr⟩ ::= Expression binary_op Expression

We call the reader's attention to the Go Language Specification [14] for the various types of `unary_op` and `binary_op`. In the following, we denote an arbitrary unary operator by $\circ_1$ and a binary one by $\circ_2$. Unary and binary expressions are checked for candidates by calling the `candidates` function on the respective operands:

$$\texttt{candidates(t, e}_g\texttt{)} = \begin{cases} \texttt{candidates(t, e}_g\texttt{')} & \texttt{if e}_g \equiv \circ_1\ \texttt{e}_g\texttt{'} \\ \texttt{candidates(t, e}_g\texttt{')} \cup \texttt{candidates(t, e}_g\texttt{")} & \texttt{if e}_g \equiv \texttt{e}_g\texttt{'} \circ_2 \texttt{e}_g\texttt{"} \end{cases}$$

This concludes all cases of the `candidates` function that deal with expressions. Now, we define the `candidates` function for statements that might entail candidates.

Statement Candidates

The Go Language Specification [14] details a wide variety of statements such as `BreakStmt`, `ContinueStmt`, `GoToStmt`, etc. For the candidate inspection, we are only interested in statements that either directly introduce new variables or allow for declaration statements in their bodies. Statements that do not introduce new variables might use candidates but these candidates would have been previously declared and thus already saved by the `candidates` function. Therefore, we limit the definition of the `candidates` function to the following kind of statements:

⟨`Statement`⟩ ::= `Declaration` | `SimpleStmt` | `Block` | `IfStmt` | `SwitchStmt` | `ForStmt` | `DeferStmt`

⟨`Declaration`⟩ ::= `VariableDecl` | `ShortVariableDecl`

⟨`SimpleStmt`⟩ ::= `ExprStmt` | `Assignment` | `ShortVariableDecl`

First, we look at declarations: Both short and regular variable declarations define one or more variables that we refer to with $\texttt{e}_g\texttt{.Variables}$. For all the newly declared variables, the `candidates` function is called in order to check whether the variable has the candidate type and should therefore be saved:

$$\texttt{candidates(t, e}_g\texttt{)} = \begin{cases} \bigcup_{\texttt{v} \in \texttt{e}_g\texttt{.Variables}} \texttt{candidates(t, v)} & \texttt{if e}_g\ \texttt{VariableDecl} \\ & \quad\ \texttt{or ShortVariableDecl} \end{cases}$$

If-, switch- and for-statements allow for variable declarations or assignments in an initial statement that is executed before the respective condition is checked. This

statement has a `SimpleStmt` type and is denoted as $\texttt{e}_g\texttt{.SimpleStmt}$. It needs to be inspected for candidates, hence, we call the `candidates` function on the simple statement for these three constructs. All other parts of an if-, switch- or for-statement are expressions that do not introduce new variables and are hence disregarded. Expression and defer statements may contain expressions such as function calls which need to be checked for candidates. This is performed with a recursive call on the (deferred) expression. If a block statement is encountered, the `candidates` function is called for each statement of this block. For assignment statements, we are only concerned with the right hand side which might introduce new candidates. The left side of an assignment will not be a new candidate since it has been previously declared in order to be now assigned a new value. Therefore, assignments are ultimately handled by recursing on the right hand side of an assignment:

$$\texttt{candidates(t, e}_g\texttt{)} = \begin{cases} \texttt{candidates(t, e}_g\texttt{.SimpleStmt)} & \texttt{if e}_g\texttt{ IfStmt, SwitchStmt or ForStmt} \\ \texttt{candidates(t, e}_g\texttt{.Expression)} & \texttt{if e}_g\texttt{ ExprStmt or DeferStmt} \\ \bigcup_{\texttt{s} \in \texttt{e}_g\texttt{.Statements}} \texttt{candidates(t, s)} & \texttt{if e}_g\texttt{ Block} \\ \bigcup_{\texttt{rhs} \in \texttt{e}_g\texttt{.Rhs}} \texttt{candidates(t, rhs)} & \texttt{if e}_g\texttt{ Assignment} \end{cases}$$

This concludes the definition of the different cases of the `candidates` function. Having decided on the heap locations that are potentially referred to in an exclusive old expression, i.e. the candidates, the next subsection discusses the declaration of lookup maps that are used to save the candidate values.

4.6.3 Lookup Map Declaration

A *lookup map* is a Go map that is used to save candidate values. A candidate might have different values at different labeled program points. Thus, in order to differentiate values of candidates at labeled program points, we need distinct lookup maps per label. A lookup map for a label `L` maps an address that represents a candidate for an expression $\texttt{e}_s$ to the candidate's value at program point `L`. The previously declared `candidates` function returns addresses corresponding to candidates. At program point `L`, these addresses are inserted in the lookup maps together with their current value. Since addresses are unique and consistent during program execution, they serve as keys to lookup values in the lookup maps.

Intuitively, we model a partial heap by storing only relevant addresses, i.e. candidates. In order to ensure type safety, we need distinct lookup maps for each type of candidate value. As explained in subsection 4.6.1, the `candidateTypes` function

returns for a given old expression a set of triples which each consist of a type, a subexpression of the given old expression and a label. The triple's type is the type of the candidate values for the subexpression. The triple's label is the label at which the old value for the subexpression needs to be looked up. For each triple returned by `candidateTypes`, a lookup map is defined using the following `map` function:

Definition 25. *Let $\mathbb{T}$ be the set of Go types, $\mathcal{S}_e$ the set of specification expressions, L the set of labels, and $\mathbb{M}$ the set of lookup maps. Then, we can define the following function*

$$map \quad : \quad \mathbb{T} \times \mathcal{S}_e \times L \rightarrow \mathbb{M}, \quad (t,\ e_s,\ L) \mapsto map_{(T,\ e_s,\ L)}$$

that declares a lookup map for a triple of candidate type, specification expression and label.

A lookup map maps addresses (i.e. candidates) of type `*T` to values (i.e. candidate values) of type `T`:

$$\texttt{map}_{(\texttt{T},\ \texttt{e}_s,\ \texttt{L})} : \texttt{*T} \rightarrow \texttt{T}$$

We would like to remind the reader of the special treatment of maps described in the previous subsection. Now, we again define a special case for map types. Since we want to be able to lookup values in an old map, we can simply define the lookup map to be an exact copy of the map at the old program point:

$$\texttt{map}_{(\texttt{T},\ \texttt{e}_s,\ \texttt{L})} : \texttt{T.key} \rightarrow \texttt{T.value} \quad \text{if T map type}$$

4.6.4 Performing Lookups

In the paragraph above, we have defined lookup maps that will be populated with candidate values for exclusive old expressions. An open question that remains is how these lookup maps are used to retrieve the correct old value that belongs to an exclusive old expression. This subsection presents the function `lookups` that returns a previously stored value for a given expression and a label at which point in the program that expression should be evaluated.

Definition 26. *Let $\mathcal{S}_e$ the set of specification expressions, L the set of labels, and $\mathcal{G}_e$ the set of Go expressions. Then, we can define the following function*

$$lookups \quad : \quad \mathcal{S}_e \times L \rightarrow \mathcal{G}_e, \quad (e_s,\ L) \mapsto e_g$$

that defines how the exclusive old value e_g of a given specification expression e_s at some label `L` is looked up. The function definition is split into the following cases

$$
lookups(e_s,\ L) = \begin{cases}
x & if\ e_s \equiv x\ \in\ \mathbb{X}\ \times\ \mathbb{L} \\
map_{(T_{e_s},\ e_s,\ L)}\ [\ lookups(e_s\text{'},\ L)\ ] & \\
\qquad\qquad if\ e_s\ \equiv\ *e_s\text{'} & \\
lookups(e_s\text{'},\ L).f & \\
\qquad\qquad if\ e_s\ \equiv\ e_s\text{'}.f & \\
lookups(e_s\text{'},\ L)\ [\ lookups(e_s\text{"},\ L)\ ] & \\
\qquad\qquad if\ e_s\ \equiv\ e_s\text{'}[e_s\text{"}],\ e_s\text{'}\ array & \\
map_{(T_{e_s},\ e_s,\ L)}\ [\ \&e_s\text{'}[0]\ +\ size(e_s\text{'}[0])\ *\ lookups(e_s\text{"},\ L)\ ] & \\
\qquad\qquad if\ e_s\ \equiv\ e_s\text{'}[e_s\text{"}],\ e_s\text{'}\ slice & \\
map_{(T_{e_s}\text{'},\ e_s,\ L)}\ [\ lookups(e_s\text{"},\ L)\ ] & \\
\qquad\qquad if\ e_s\ \equiv\ e_s\text{'}[e_s\text{"}],\ e_s\text{'}\ map &
\end{cases}
$$

The first case of the lookup function defines that no lookup occurs for constants, literals and exclusive variables since they always evaluate to their current value. A dereference results in a lookup in the map belonging to the type of the dereference expression. The lookup key is the result of the recursive call of the function `lookup` on the dereferenced pointer in order to allow for nested lookups. Field accesses are performed on the lookup result of the structure. For index expressions, we define cases for arrays, slices and maps separately. If an index expressions on an array is encountered, we call the `lookups` function on both the array and the index. The result is an index expression constructed from both these lookups. For index expressions on slices, we first recall that for slices candidates are other slices as well as arrays since slices are constructed from arrays. We further recall that every member of a candidate slice or array was saved as a candidate value instead of saving actual slices. Therefore, we need to directly lookup the candidate value at the given index. Since the index of a slice might be shifted in regard to the index of the underlying array, we use pointer arithmetic to find the correct index for the lookup. The index is the address of the first slice entry shifted by the provided index which we receive using a recursive call of the lookups function. The provided index is multiplied by `size(`e_s`'[0])` because a shift has to occur in steps equal to the size of the slice elements. For index expressions on maps, we recollect that the corresponding lookup map is simply a copy of the old map. Therefore, the key of the index expression is looked up and used as a key in the lookup map which returns the corresponding old value.

4.6.5 Exclusive Old Algorithm

The functions `candidateTypes`, `candidates`, `map` and `lookups` declared in the last four subsections provide the functionality that we need to define an algorithm that

performs the runtime check generation of exclusive old expressions. This algorithm describes how the candidate and lookup maps are filled in order to be able to perform valid lookups. We call the algorithm the *exclusive old algorithm*. We execute the exclusive old algortihm once on every function of the program for which runtime checks are generated. The exclusive old algorithm is split into three parts, namely the initialization of auxiliary data structures, the handling of parameters and return parameters, and the main part that traverses the body of a function. We thus start with the first part of the exclusive old algorithm that shows how candidate sets and lookup maps for all exclusive old expressions that are used in the function's body are initialized:

Exclusive Old Algorithm. Part 1: Initialization of candidate sets and lookup maps (Subsequent parts of the algorithm will use these data structures)

Data: Function declaration `f`
Result: Set of candidate sets $\mathbb{C}$, set of lookup maps $\mathbb{M}$
$\mathbb{C} \leftarrow \emptyset$;
$\mathbb{M} \leftarrow \emptyset$;
for `old[L](e)` used in the body of `f` **do**
  for (t, e, L) $\in$ `candidateTypes(e)` **do**
    $\mathbb{M} \leftarrow \mathbb{M} \cup \{\texttt{map}_{\texttt{(t, e, L)}}\}$;
    if $\neg\exists\, cs_t \in \mathbb{C}$ **then**
      $cs_{\texttt{t}} \leftarrow \emptyset$;
      $\mathbb{C} \leftarrow \mathbb{C} \cup \{cs_{\texttt{t}}\}$;

The set $\mathbb{C}$ holds all candidate sets and the set $\mathbb{M}$ all lookup maps. Both sets $\mathbb{C}$ and $\mathbb{M}$ are initialized as the empty set. A candidate set for candidates of type t is denoted as $cs_{\texttt{t}}$. All candidate sets are also initially empty.

The exclusive old algorithm starts by iterating over all exclusive old expressions used in the body of the given function `f`. For each old expression, it computes the set of candidate types and iterates over all triples that are returned by `candidateTypes`. For each triple, a lookup map is defined and included into the set of lookup maps $\mathbb{M}$. For the returned candidate type, we check whether a corresponding initial candidate set already exists in $\mathbb{C}$. We perform this check since candidate sets are independent of the labels; we need only one candidate set per unique candidate type. I.e. if the triples `(t, e, L)` and `(t, e', L')` are returned by a call to `candidateTypes`, we declare two lookup maps $\texttt{map}_{\texttt{(t, e, L)}}$ and $\texttt{map}_{\texttt{(t, e', L')}}$ but only a single candidate set $cs_{\texttt{t}}$.

After the initialization of the lookup maps and candidate sets, the exclusive old algorithm 2 first checks whether any parameters or return parameters of `f` are candidates for some old expression.

Exclusive Old Algorithm. Part 2: Filling candidate sets for (return) parameters

Data: Function declaration `f`, set of initialized candidate sets $\mathbb{C}$
Result: Set of updated candidate sets $\mathbb{C}$
for $p \in$ `f.parameters` $\cup$ `f.returnParameters` **do**
  for $cs_t \in \mathbb{C}$ **do**
    $cs_t \leftarrow cs_t \cup$ `candidates`$\{p, t\}$;

For each parameter of `f` we iterate over all candidate sets and call the `candidates` function. Input to the function call to `candidates` is the parameter and the type of the current candidate set. The returned addresses of candidate values are then added to the current candidate set. The same procedure is applied to return parameters as well.

Exclusive Old Algorithm. Part 3: Filling of candidate sets and lookup maps

Data: Function declaration `f`, set of candidate sets $\mathbb{C}$, set of lookup maps $\mathbb{M}$
Result: Set of complete candidate sets $\mathbb{C}$, set of filled lookup maps $\mathbb{M}$
for Statement $s \in$ body of `f` **do**
  if $s \equiv$ `L` where `L` is label **then**
    for $cs_t \in \mathbb{C}$ **do**
      for $\texttt{map}_{(t',e,L')} \in \mathbb{M}$ **do**
        if $t' \equiv t \wedge L' \equiv L$ **then**
          for `addr` $\in cs_t$ **do**
            if `addr` is map pointer **then**
              for $k, v \in$ `*addr` **do**
                $\texttt{map}_{(t',e,L')}[k] \leftarrow v$;
            else
              $\texttt{map}_{(t',e,L')}$[`addr`] $\leftarrow$ `*addr`;
  else
    for $cs_t \in \mathbb{C}$ **do**
      $cs_t \leftarrow cs_t \cup$ `candidates`$\{n, t\}$;

The last part of the exclusive old algorithm performs a traversal over all statements in the body of the function `f`. For each statement that is encountered during the traversal, all candidate expressions are added to the corresponding candidate set. When encountering a label that is used in an old expression, all lookup maps for that label are filled. Filling a lookup map means that every address in the suitable candidate set is mapped to its current value and this mapping is inserted into the

lookup map. The current value is the value that is obtained by dereferencing the address at the label. An exceptional case is defined for maps as was discussed in Subsection 4.6.3. Here, all keys and their old values are copied by iterating over all entries of the dereferenced map pointer.

4.6.6 Encoding

After the exclusive old algorithm has been executed on a function that contains some exclusive old expression `e`, the lookup map for `e` contains all potential old values of this expression at runtime. Therefore, the old expression `e` can be translated to Go code by replacing it with a lookup of its old value in the corresponding lookup map.

Encoding 14. *For the runtime check generation of exclusive old expressions we have the following encoding:*

$$\langle\!| \mathit{old[L]}(\langle e \rangle) |\!\rangle \rightsquigarrow \mathit{lookups}(\langle e \rangle, \mathit{L})$$

The encoding relies on a previous execution of the exclusive old algorithm described in Section 4.6.5 which ensures that all auxiliary data structures will be initialized and filled at runtime.

With this encoding we finish the description of the runtime check generation for exclusive old expressions. The next two subsections discuss how exclusive old expressions are handled in the implementation of GoRAC and give some remarks on the usage of exclusive old expressions with nested expressions and with pointer types.

4.6.7 Implementation

The current implementation of GoRAC provides partial support of exclusive old expressions. In this section, we outline which usages of exclusive old expressions are supported and how the support can be extended.

We first observe that exclusive variables used in postconditions behave analogously to shared variables in postconditions. This is due to the fact that parameters in postconditions relate to values at the beginning of a function. A shared variable in old evaluates to the value at the corresponding label: In postconditions, this is simply the value of the parameter. An exclusive variable in old evaluates to its current value: In postconditions, this is also the value of the parameter. Since GoRAC provides full support of shared old expressions, exclusive old expressions used in postconditions are supported by treating them like shared old expressions.

In any other specification annotation, the usage of exclusive old expressions is currently not possible because the exclusive old algorithm wasn't implemented in its entirety due to time reasons. Nevertheless, we want to provide some technical details for the existing partial implementation as well as future parts to come. The given details show that the overall concept of the exclusive old algorithm was assessed and that we were able to already establish certain guidelines for a future complete implementation of the algorithm.

Slice Lookups: The lookup function that was introduced in Section 4.6.4 defines that slice lookups are performed using pointer arithmetics: the lookup map uses addresses of positions in the slice to refer to the old values at these positions. When a value at an index is looked up, the address for the lookup is calculated by adding the index to the address of the slice's first entry. In order to perform pointer arithmetic as required by slice lookups, we need to use Go's `unsafe` package. We have sketched a small example in Figure 4.11 that demonstrates the necessary pointer arithmetic to calculate a slice element's address. The original code that includes the specification is displayed on top. The code at the bottom depicts the implementation of the

```
func sliceLookup() {
    a := [5]int{1, 2, 3, 4, 5}
    s := a[1:] //@ exclusive: s
    //@ L:
    a[3] = 0
    //@ assert old[L](s[2]) == 4
    //@ assert s[2] == 0
    _ = s // avoid complains about unused variable s
}
```

```
func sliceLookup() {
    candidates := map[*int]bool{}
    lookup := map[unsafe.Pointer]int{}
    a := [5]int{1, 2, 3, 4, 5}
    for i := range a {
        candidates[&a[i]] = true
    }
    s := a[1:]
    //@ L:
    for c := range candidates {
        lookup[unsafe.Pointer(reflect.ValueOf(c).Pointer())] = *c
    }
    a[3] = 0
    if !(lookup[unsafe.Pointer(reflect.ValueOf(&s[0]).Pointer() +
    uintptr(unsafe.Sizeof(s[0]) * 2))]) == 4 {
        panic("Assertion at line 6 violated.")
    }
    if !(s[2] == 0) {
        panic("Assertion at line 7 violated.")
    }
}
```

Fig. 4.11: Implementation of a slice lookup

runtime checks using the `unsafe` package for the pointer arithmetic during the slice lookup.

Map Lookups: As shown in the definition of the `candidates` function and in Algorithm 3, we handle map lookups using pointers to maps. This also requires the use of the `unsafe` package which we demonstrate in Figure 4.12. Some more work will be needed for map composite literals since Go does not allow to take the address of composite literals. This could be overcome by first assigning the composite literal to a variable and then taking the address of this variable instead.

```
func mapLookup() {
    m := map[int]int{42:42}
    //@ L
    m = map[int]int{42:1337}
    //@ assert old[L](m[42]) == 42
    _ = m // avoid complains about unused variable m
}
```

```
func mapLookup() {
    candidates := map[unsafe.Pointer]bool{}
    lookup := map[int]int{}
    m := map[int]int{42:42}
    candidates[unsafe.Pointer(reflect.ValueOf(&m).Pointer())] = true
    //@ L
    for c := range candidates {
        for key, value := range *(*map[int]int)(c)) {
            lookup[key] = value
        }
    }
    m = map[int]int{42:1337}
    if !(lookup[42] == 42) {
        panic("Assertion ... at line 5 violated.")
    }
    _ = m
}
```

Fig. 4.12: Implementation of a map lookup

4.7 Remarks on Old Expressions

We conclude the old expressions part of the chapter on runtime check generation with two remarks: The first one focuses on the correct usage of pointer types in both shared and exclusive old expressions. The second remark concerns a limitation that is placed on nesting shared old expressions.

4.7.1 Pointer Types in Old Expressions

It is important to note that the old value of an expression whose type is a pointer type is a copy of the pointer itself. We do not copy the object the pointer refers to. Hence, if an object is changed in between a label and an old expression (with

```
type foo struct {
    bar int
}

//@ require foo.bar == 42
func changeBar(x *foo) { // x either exclusive or shared, same outcome
    x.bar = 1337
    //@ assert old(x).bar == 1337
    //@ assert old(x.bar) == 42
}
```

Fig. 4.13: Pointer types used in old expressions

that label) that reasons about a pointer to the object, these changes will be visible through old.

The example from Figure 4.13 demonstrates in the first assertion the semantics of pointer types in old expressions as explained above. In the beginning, the pointer `x` passed to the function points to a `foo` object with field `bar` equal to 42. Since only the pointer, i.e. the address it holds, is copied when computing the old value of `x` and not the object itself, the old value of `x` points to the same object as `x`. Since `x.bar` is changed before the first assertion, `old(x)` still points to the new changed object and therefore the assignment is observable. As shown in the second assertion, if one wants to obtain to the old value of `x.bar`, it is required to write `old(x.bar)` instead of `old(x).bar`.

4.7.2 Restriction on Nested Shared Old Expressions

Consider the code provided in Figure 4.14. To be able to correctly evaluate the old expression

`old[L0](*old[L1](*x))`

that is part of the assertion in Figure 4.14, we would first need to evaluate the inner old expression `old[L1](*x)`. Only afterwards, we could proceed with the evaluation of the outer old expression `old[L0](...)`. However, for runtime checking this

```
//@ requires **x == 1 && **y == 0
func nestedOld(x, y **int) { //@ shared: x, y
    //@ label L0
    *x = *y
    **y = 2
    //@ label L1
    //@ assert old[L0](*old[L1](*x)) == 2
}
```

Fig. 4.14: Nested old expressions

proves to be problematic since label `L1` is reached after label `L0`, i.e. at `L0` the old expression `old[L1](*x)` has not yet been evaluated. Thus, we face the problem of evaluating an outer old expression `old[L0](...)` that relies on the future evaluation of its operand.

Our current support of shared old expressions cannot handle this problem. Therefore, we impose the restriction that for nested expressions, any inner label has to occur earlier in terms of control flow of the program before any outer label. In other words, inner labels have to already been reached when reaching an outer label.

This situation is similar to the one that occurs for exclusive old expressions. Hence, we could use a similar approach to the exclusive old algorithm to overcome the presented restriction. Besides the fact that the exclusive old algorithm has not been implemented yet, such a solution for nested old expressions would also yield space and execution time disadvantages.

Note that the problem outlined above would also occur if we introduced a specification construct `now` with the semantics that calling `now` on a shared variable in an old expression results in this variable to be evaluated to its current value. I.e. `now` makes a shared variable behave like an exclusive one.

4.8 Permissions

Recall that GoRAC permits the declaration of access permissions on pointers, slices, maps, reference expressions, and implicit field accesses, i.e. field access on struct pointers. The following encodings show how each valid type of access permission is converted to Go code. Since we do not keep track of actual permissions at runtime, we over-approximate the access permissions by checking whether the pointer refers to a valid object. I.e. the specified access permission fails for null pointers.

Encoding 15. *Let* e *be a pointer, slice or map,* e_1 *be a struct pointer with field* e_2 *and* e_3 *be an addressable expression. Then, we can encode access permissions stated on these expressions as follows:*

$$\langle\!|\, acc(\langle e \rangle) \,|\!\rangle \rightsquigarrow \langle\!|\, \langle e \rangle \,|\!\rangle \texttt{ != nil}$$

$$\langle\!|\, acc(\langle e_1 \rangle.\langle e_2 \rangle) \,|\!\rangle \rightsquigarrow \langle\!|\, \langle e_1 \rangle \,|\!\rangle \texttt{ != nil}$$

$$\langle\!|\, acc(\&\langle e_3 \rangle) \,|\!\rangle \rightsquigarrow \&\langle\!|\, \langle e_3 \rangle \,|\!\rangle \texttt{ != nil}$$

The first line of the encoding shows that identifiers referring to pointers are simply encoded by checking whether the pointer is not equal to nil. The second line of the encoding shows that the same check is performed for structures of dot notations. The structure must also be a pointer type. Note that the dot notation might be an implicit dereference of the pointer. The third line of the encoding might seem a little bit redundant since taking the address of any expression will always yield a value that is not nil. Therefore, we could encode the access permission into a simple `true` statement. However, in order to allow the type checker that is run on the generated code to check whether the reference expression is valid, we cannot simply replace this check by `true`. This has lead to the decision of encoding an access permission for a reference with a regular check for nil.

The code in Figure 4.15 demonstrates the runtime check generation of access permission checks. The function shown at the top of the figure adds the value of an integer pointer to the field of a struct. It has a struct pointer as receiver and an integer pointer as a parameter, both of which are specified to be accessible. The generated code at the bottom shows that the runtime checks test whether both pointers are not equal to `nil`. If so, the parameters are deemed accessible, in the other case, the program terminates with a panic.

```
type foo struct {
    bar int
}

//@ requires acc(x) && acc(f.bar)
func (f *foo) addPtr(x *int) {
    f.bar += *x
}
```

```
func (f *foo) addPtr(x *int) {
    if !(x != nil && f != nil) {
        panic("Precondition violated.")
    }
    f.bar += *x
}
```

Fig. 4.15: Runtime checks for access permissions

4.9 Predicates

The runtime check generation of predicate calls relies on encodings of both the predicate call itself as well as the declaration of the predicate that was called. We first address the encoding of predicate declarations and then continue with the runtime check generation of predicate calls.

4.9.1 Predicate Declarations

A predicate declaration is translated into a function definition. The resulting function is named like the predicate. This is also the reason why a predicate needs to have a unique name as described in Section 3.6 such that we can distinguish generated functions from existing functions. The generated function returns the encoded assertion.

Encoding 16. *A predicate named ⟨P⟩ with parameters ⟨X⟩ and assertion ⟨a⟩ is encoded in Go code as follows:*

$$\llbracket \texttt{predicate}\ \langle P\rangle\ \texttt{(}\langle X\rangle\texttt{)}\ \texttt{\{}\ \langle a\rangle\ \texttt{\}} \rrbracket \rightsquigarrow$$

```
func ⟨P⟩ ( ⦇ ⟨X⟩ ⦈ ) bool {
    return ⦇ ⟨a⟩ ⦈
}
```

The set ⟨X⟩ contains parameter declarations that are tuples consisting of a parameter name ⟨x⟩ and a parameter type ⟨T⟩. The encoding ⦇ ⟨X⟩ ⦈ encodes all parameter declarations by translating the specification identifiers ⟨x⟩ and ⟨T⟩ into identical Go identifiers `x` and `T`.

Since all named functions in Go need to be declared in the root scope of a program, i.e. named function definitions cannot be nested, we include the generated predicate function also into the root scope. With this remark, we conclude the runtime check generation of predicate declarations. The next subsection proceeds with runtime checking of predicate calls.

4.9.2 Predicate Calls

Before a predicate call is converted to Go code, the corresponding predicate declaration is encoded as a function and included into the program as described in the previous subsection. Encoding a predicate call then simply corresponds calling the corresponding function:

Encoding 17. *A predicate call of some predicate with unique name ⟨P⟩ is translated to Go code using the following encoding:*

$$\llparenthesis\ \langle P\rangle\ \texttt{(}\ \langle e\rangle\texttt{*)}\ \rrparenthesis \rightsquigarrow \langle P\rangle\ \texttt{(}\ \texttt{paramValues(}\langle e\rangle\texttt{*)}\ \texttt{)}$$

The encoding shows that values of the parameters passed into the predicate call are translated using a helper function:

Definition 27. *Let $\mathcal{S}_e$ denote the set of specification expressions and $\mathcal{G}_e$ the set of Go expressions as described in the Go Language Specification [14]. Then, we can define the function*

$$\texttt{paramValues}\colon \mathcal{S}_e^* \to \mathcal{G}_e^*,\ \langle e \rangle * \mapsto paramValues(\langle e \rangle *)$$

that maps a list of parameter values $\langle e \rangle$ from a predicate call to a Go identifier list expression [14] of parameter values for the corresponding predicate function using the following recursive definition:*

$$\texttt{paramValues}(\langle e \rangle *) = \texttt{paramValues}(\langle e_1 \rangle, \ldots, \langle e_n \rangle) =$$
$$\begin{cases} \{\!| \langle e_1 \rangle |\!\} & \texttt{if } n \equiv 1 \\ \{\!| \langle e_1 \rangle |\!\}, \texttt{paramValues}(\langle e_2 \rangle, \ldots, \langle e_n \rangle) & \texttt{otherwise} \end{cases}$$

Note that during the runtime check generation of predicate calls, GoRAC checks whether two different requirements are fulfilled: First, a predicate declaration needs to be in scope of a corresponding predicate call. And second, the values of predicate call parameters need to have the correct types as stated in the corresponding declarations.

```
//@ predicate sorted(nums []int) {
//@     forall i, j int :: i in range nums && 0 <= j < i
//@                        ==> nums[j] <= nums[i]
//@ }

//@ requires sorted(x)
func maximum(x []int) (max int) {
    return x[len(x) - 1]
}
```

```
func sorted(nums []int) bool {
    return func() {
        for i := range nums {
            for j := 0; j < i; j++ {
                if !(nums[j] <= nums[i]) {
                    return false
                }
            }
        }
        return true;
    }()
}

func maximum(x []int) (max int) {
    if !(sorted(x)) {
        panic("Precondition violated")
    }
    return x[len(x) - 1]
}
```

Fig. 4.16: Runtime check generation of predicate calls

Figure 4.16 shows an example predicate known from Section 3.6 that checks whether a given integer slice is sorted in increasing order. The predicate is called in a precondition for the maximum function depicted at the top of the figure. Since the predicate is defined in scope of the predicate call and its name does not collide with any existing function name, the runtime check generation succeeds. The resulting code at the bottom shows the generated function definition for the predicate and the predicate call encoded as a function call as part of the precondition runtime check.

4.10 Purity

Purity annotations as defined in Section 3.7 express constraints on the syntax of a function. A function satisfies a purity annotation if it has exactly one return parameter, its body consists of only a single return statement returning a pure expression, and any assertion of postcondition for the function is a pure expression. A function annotated as being pure has to be pure in all potential executions. Since checking such a property at runtime would be difficult (or even impossible), we decided to syntactically check pureness. With this, we follow the approach of the Go verifier Gobra [2].

Purity checks are performed in two steps: First, GoRAC checks whether the syntax of any function annotated as being pure fulfills the purity requirements given in Section 3.7. Then, for all function calls occurring in specification GoRAC checks whether the called function is annotated as pure.

The checks are performed during the execution of GoRAC. If a function is declared as being pure but does not abide by the constraints described in Section 3.7, or a function is called which is not pure, GoRAC aborts the runtime check generation and gives an error to the user. When all purity checks performed by GoRAC are successful, the purity annotations are removed. Thus, no runtime check is generated and no encoding needs to be declared for purity annotations.

The explanation of purity checks concludes the chapter on runtime check generation. The chapter illustrated how specification annotations are translated to Go code and included into a given program such that the conditions they express are validated during program execution. The next chapter details GoRAC's implementation and gives further details on how the runtime check generation is realized.

5 Implementation

> *If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea.*
>
> **— Tim Peters**
> Distinguished Software Engineer

Go Runtime Assertion Checker (GoRAC) is a command line tool implemented in Go [18]. It consists of two Go packages: a specification parser and a framework that handles user input, the overall process of generating runtime assertion checks and the tool output. Runtime checks are generated as described in Chapter 4. This chapter is structured as follows: First, an overview of the framework's implementation is given in Section 5.1. Then, Section 5.2 details implementation specific changes to the encodings as given in Chapter 4. The tool's usage is explain in Section 5.3.

5.1 Overview

A visualization of GoRAC's workflow is given in Figure 5.1. On the left-hand side, a Go source file that is annotated with specification is input to GoRAC. It is converted into a file containing the original sources including runtime assertion checks (RACs) for the provided specification, as shown on the right-hand side. Input and output

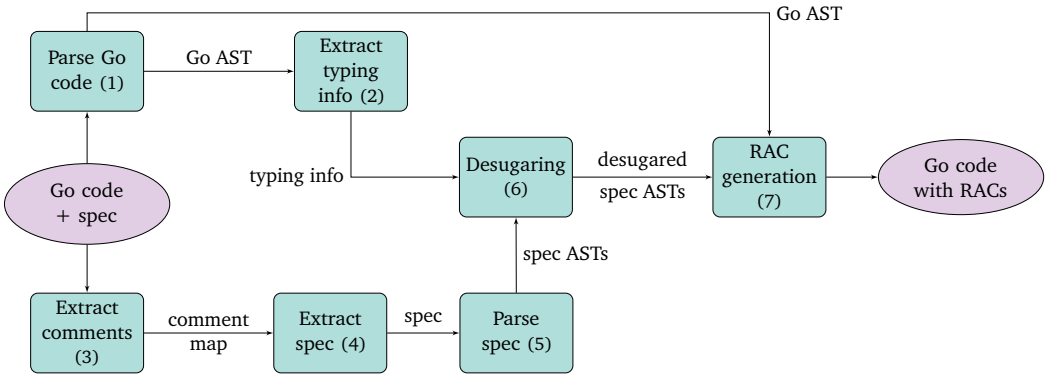

Fig. 5.1: Overview of the GoRAC's workflow
(AST = Abstract Syntax Tree, RACs = Runtime Assertion Checks)

are depicted in purple ellipses, processes in green boxes. The arrows indicate the order in which process output is combined for further steps.

1. First, the Go code is parsed into an abstract syntax tree (AST) using the Go package `go/ast` [15].
2. From the Go AST, we obtain typing information using the package `go/types` [17].
3. Additionally, we extract all comments with the help of the package `go/parser` [16] and receive a comment map. A comment map holds for every comment in the provided Go file a reference to an AST node that belongs to the comment.
4. Following the figure, comments are then filtered by being specification comments or not.
5. Then, the specification parser is used to parse each specification annotation into a specification AST.
6. The specification ASTs and the typing information are combined during the desugaring process. The results are desugared specification ASTs. We need the typing information to correctly transform the specification ASTs into runtime assertion checks.
7. Each desugared specification AST is translated into a runtime assertion check by the runtime assertion check generator. This part of the framework implements the encodings of the specification constructs as explained in Chapter 4. The runtime assertion checks are inserted into the original Go AST.
8. Finally, the modified Go AST, that now includes the runtime assertion checks, is written to a file that can be compiled.

We explain two processes of the depicted workflow in Figure 5.1 in more detail: the process of parsing specification and the process of generating the runtime assertion checks.

Specification annotations are parsed by a *specification parser*. The specification parser parses each specification comment, containing one or multiple specification clauses, into an AST. The parser is implemented using ANother Tool for Language Recognition (ANTLR) [35]. ANTLR is a parser generator for LL(k)-parsers [36], i.e. left-to-right parsers that use k tokens of lookahead. The parser for the GoRAC specification language is constructed using a grammar and visitor functions. The grammar can be reused in other projects, even in projects written in other programming languages than Go since it is agnostic to the programming language. To allow for further reuse, we made the design choice to place the specification parser in a separate Go package and import it in the GoRAC implementation.

The runtime check generation is performed by a GoRAC component called *runtime check generator*. The runtime check generator receives as input desugared specification ASTs and the Go AST of the original program. The generator performs several checks before creating and inserting the runtime checks for the specification ASTs into the Go AST, as shown in Figure 5.2. First, purity checks are performed: Every function annotated as pure is checked for pureness as described in Section 4.10. For each function call that is part of a specification condition, it is additionally checked whether the corresponding function declaration is annotated as pure. Second, for all quantifiers it is checked whether every non-boolean quantified variable is bound by some domain. Third, all old expressions are extracted from specification assertions. We check for each exclusive old expression whether its use is supported or not. Details on the correct placement of old expressions are given in Section **??**. If it is supported, we encode the old expression as described in Section 4.5. Fourth, we perform predicate extraction, i.e. for every predicate call, we include the respective predicate as a function declaration in the provided Go AST. The fifth step is the translation of the specification ASTs into Go AST nodes. These translations have been described in the form of encodings in Chapter 4. In the last step, the resulting nodes are inserted into the Go AST. The resulting tree is an abstract syntax representation of the original program with runtime assertion checks for all specification annotations.

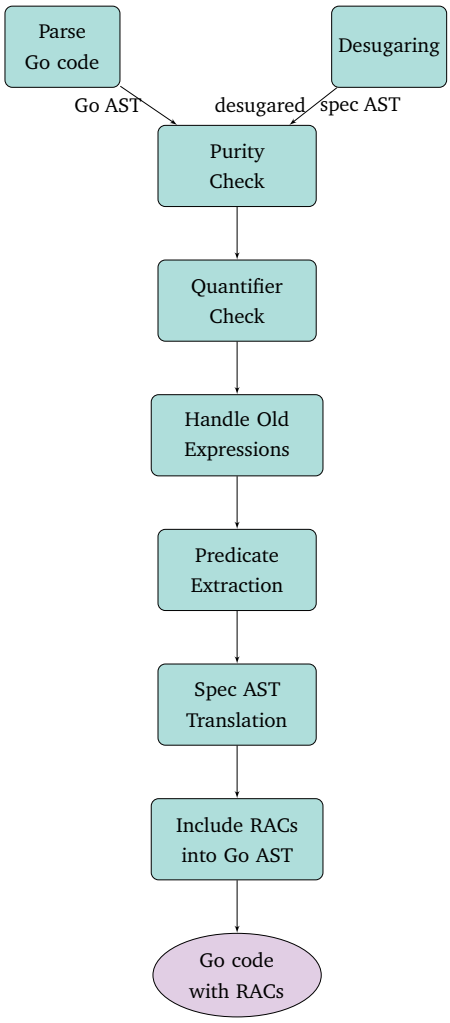

Fig. 5.2: Runtime check generation process (AST = Abstract Syntax Tree, RACs = Runtime Assertion Checks

5.2 Wrapper

The encoding of specification constructs, as described in Chapter 4, states that e.g. an assertion `//@ assert a[0] > 42` is encoded as

```
if !(a[0] > 42) {
    panic("Assertion 'x > 42' violated.")
}
```

If the assertion condition is violated, we expect the program to fail with the message `"Assertion 'x > 42' violated."` However, the encoding does not meet these expectation if the array `a` is empty. The program will crash when accessing the element at index zero and result in the following error `"runtime error: index out of range [0] with length 0"`. This message neither relates to the assertion that does not hold, nor will it give information about the line of the original specification that caused this error. Therefore, we extend the encoding of runtime assertion checks in the implementation with a wrapper. The wrapper catches all errors that occur while evaluating the runtime assertion checks and extends an error message by line information and the violated specification.

For the example above, the actual produced code by GoRAC is shown in Figure 5.3. This wrapping ensures for the example given at the beginning of this subsection that the following message will be displayed to the user in case a panic occurs while evaluating the assertion: `"Line 1337, Specification 'assert x > 42': runtime error: index out of range [0] with length 0"`. Hence, the user is able to quickly locate the panic's origin and fix it.

```
func() {
    defer func() {
        if err := recover(); err != nil {
            panic(fmt.Sprintf("Line 1337,
            Specification 'assert x > 42': %v", err))
        }
    }()
    if !(a[0] > 42) {
        panic("Assertion 'x > 42' violated.")
    }
}()
```

Fig. 5.3: Implementation of a runtime check that catches errors and adds line information

5.3 Tool

GoRAC is a command line tool and its operation can be configured via various command line flags. By default, GoRAC produces checks for all specification annotations. We can configure GoRAC to produce only runtime assertion checks for certain types of specification clauses with the command line flags `-generatePrePostChecks`, `-generateInvariantsChecks`, and `-generateAssertionChecks`. In its default configuration, the resulting program is printed to a file that has the same name as

the input file extended with the suffix `_rac.go` The flag `-outputFile=<filename>` instructs GoRAC to write the resulting program to a file with the given name. Further configuration options as well as installation and usage instructions can be found in GoRAC's readme [28]. GoRAC 's implementation and the specification parser package are tested with a unit test suite that provides a statement coverage of approximately 88%. Code that is not covered by unit tests mostly consists of getter methods for struct types and automatically generated code of the specification parser by ANTLR.

6 Test Input Generation

> *Program testing can be a very effective way to show the presence of bugs but is hopelessly inadequate for showing their absence.*
>
> **— Edsger Wybe Dijkstra**
> Computer Science Pioneer

When GoRAC is executed on a program that includes specification annotations, it generates a file containing the code of the program enhanced with runtime assertion checks for the specification. The program needs to be executed to determine whether the runtime assertion checks hold, i.e. whether the program satisfies its specification. A single execution that passes all runtime checks guarantees that the specified properties are satisfied for exactly the input that was given for the execution. However, these guarantees cannot be generalized for executions of the program with other inputs. Therefore, the program should be run with a wide variety of different inputs. Manually devising a wide variety of program inputs requires a lot of time and effort. This effort can be reduced by automatically generating test inputs to execute the program with.

A technique that is used for test input generation is called *fuzzing*. During fuzzy testing, a program's robustness is measured by running it with random and potentially malformed inputs and thereby discovering crashes or invalid program states [22]. In this chapter we will assess different test input generation strategies for GoRAC that are based on fuzzing. We will first outline several fuzzy testing approaches in Section 6.1 and address a limitation of fuzzing when dealing with preconditions. Different solutions to overcome this limitation are discussed in Section 6.2. The chapter concludes with the explanation of a prototype implementation of one of these solutions in Section 6.3.

6.1 Fuzzing Approaches

In order to check whether a program satisfies its runtime assertion checks for different executions, we start by testing the program using existing fuzzy testing packages

for Go programs. The package *go-fuzz* is a coverage-guided fuzzing solution for testing of Go packages [43]. It generates various inputs for a given Go program in an infinite loop and aims to achieve a complete statement coverage of a program. In order to differentiate between input that increases coverage and input that does not provide relevant information, the package requires a user to write a function called `Fuzz`. Based on the output of the `Fuzz` function, the fuzzer increases or decreases priority of a given input during subsequent fuzzing. Additionally, initial input for the fuzzer called a *corpus* needs to be provided in byte format.

The requirements, that a go-fuzz user needs to provide a byte encoding of arbitrary program inputs for the corpus and an implementation of the `Fuzz` function, substantially decrease usability of the tool for programmers without knowledge of fuzzing techniques. Therefore, we attempt to automate the approach using the package *go-fuzz* [40]. (Note that *go-fuzz* is different from *gofuzz*.) The gofuzz package generates initial input which can be used during fuzzy testing a program with go-fuzz.

However, the combination of go-fuzz and gofuzz leads to further complications: Using initial input created by gofuzz, most tests that go-fuzz performs fail the runtime checks for preconditions. E.g. for the binary search algorithm that is discussed in Section 1.1, go-fuzz in combination with gofuzz is not able to generate a valid input array that is sorted. Executions that terminate due to violated preconditions do not provide any information about subsequent runtime checks. Thus, the test that are generated with the combination of gofuzz and go-fuzz are not effective. We instead seek to generate a great range of different test inputs for which the precondition holds.

6.2 Handling Preconditions

Given a precondition that reasons about several parameters, we want to generate values for the parameters such that the precondition is satisfied. This problem can be formulated as an SMT (Satisfiability Modulo Theories) [5] problem. We can use an SMT solver to generate a model for the precondition, i.e. an assignment for the parameters that satisfies the precondition.

The formulation of a precondition as an SMT problem entails an encoding of all variables of the precondition into first-order logic formulas. Since GoRAC preconditions reason about parameters that can be arbitrary Golang objects, we require an encoding of Go data structures like arrays and structs into first-order logic formulas. As such encodings are quite complex, we aim to reuse the SMT encodings of the Gobra verifier. This is possible since we defined our specification language to be

close to Gobra's specification language. For a non-satisfiable SMT formula, Gobra also supports counter example generation using the SMT solver Z3 [10]. Thus, we can employ Gobra to get a model for a given precondition as follows: We write a function with `true` as its precondition. The function's body contains an assertion that holds the negated precondition for which we want to compute a model. Gobra then encodes the precondition into SMT format and generates a counter example for the negated precondition using Z3. This counter example is a satisfying assignment, i.e. a model, for the original (non-negated) precondition.

We have implemented a prototype version of this approach in the remaining time for this thesis. The prototype is restricted to preconditions that reason about integer variables, which significantly facilitates the SMT encoding. Hence the integration of Gobra with GoRAC is omitted and the SMT solver Z3 is directly included into GoRAC's fuzzing module.

6.3 Combining SMT Solving and Fuzzing

We implement test input generation for functions with integer preconditions using a combination of SMT solving and fuzzing. Test input for parameters bound by preconditions is generated using an SMT solver while all remaining parameters receive random values obtained by a fuzzer. Figure 6.1 outlines the combination of the two techniques in order to automatically test a function with runtime checks. Input is a file generated by GoRAC, i.e. a Go program with runtime assertion checks, and the number of desired tests `n` that should be run on the given program. The input file consists of several Go functions; the figure depicts the test generation process for a single function: First, the parameters and preconditions of the function are extracted. The SMT solver Z3 generates models for the preconditions. The values of the parameters from these models are assigned to the parameters.

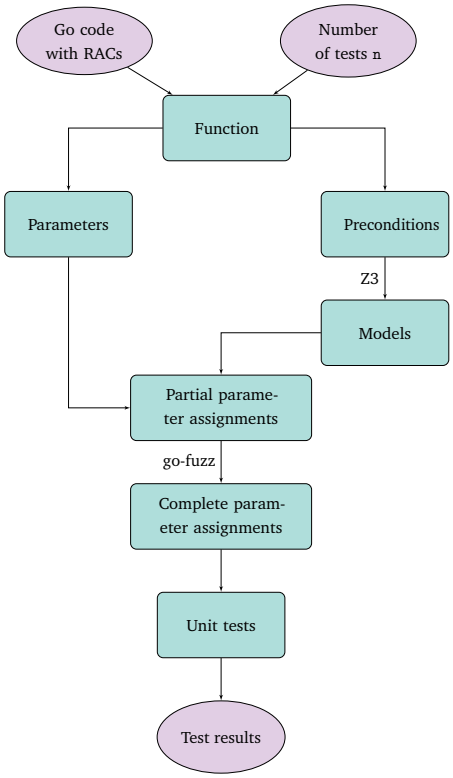

Fig. 6.1: Test input generation process (RACs = Runtime Assertion Checks)

This results in partial parameter assignments since parameters that are not part of a precondition have not been assigned a value yet. Then, gofuzz is used to generate random values for all remaining parameters. In this step, the models and random assignments are further combined such that $\mathtt{n}$ complete parameter assignments are obtained. For instance, the same model is combined with different fuzzy values of remaining parameters. From each complete parameter assignment, we construct a unit test for the function. Finally, all generated tests are executed. The test results exhibit whether or not the runtime checks of the input program hold for the generated inputs.

It remains to explain how multiple models are computed for the preconditions of a function. Intuitively, we generate models iteratively and, for each model that is obtained, we add a restriction that the same model should not be computed in future steps. Algorithm 4 describes the model computation in detail:

Algorithm 4: Computing models for preconditions

Data: Conjuction of preconditions $\mathtt{c}$, number of desired tests $\mathtt{n}$
Result: Set of models M for the conjunction of preconditions $\mathtt{c}$

$\mathtt{c}_{\mathtt{SMT}} = \mathtt{Z3.encode(c)};$
$M = \emptyset;$
$\mathtt{i} = 0;$
while $\mathtt{Z3.sat}(\mathtt{c}_{\mathtt{SMT}}) \wedge \mathtt{i} < \mathtt{n}$ **do**
  $\mathtt{m} = \mathtt{Z3.model}(\mathtt{c}_{\mathtt{SMT}}) = \{(\mathtt{x}_1, \mathtt{v}_1), \ldots, (\mathtt{x}_k, \mathtt{v}_k)\};$
  $M = M \cup \{\mathtt{m}\};$
  $\mathtt{c}_{\mathtt{SMT}} = \mathtt{c}_{\mathtt{SMT}} \wedge \bigvee_{(\mathtt{x},\ \mathtt{v}) \in \mathtt{m}} \mathtt{x} \not\equiv \mathtt{v};$
  $\mathtt{i}{+}{+};$

Input to the algorithm is a conjunction of all preconditions for the function under test. The first step of the algorithm is the encoding of the conjunction into SMT format. I.e. all Go integer variables are translated into Z3 variables, and all arithmetical, logical and relational unary and binary operations are reconstructed using respective Z3 operators. The set that will contain the computed models is initialized with the empty set and a counter to keep track of how many models have been computed is declared. After initialization, the algorithm starts by iteratively computing models in a while loop. Every model that is computed is added to the set of models which will be returned. For all variable assignments, we disjoin a condition to the precondition conjunction which states that a variable should not be equal to it's assigned value of the latest model. This disjunction ensures that every iteration of the loop yields a different model. The loop terminates if either the (modified) precondition conjunction is not satisfiable or if we have generated enough models for

the desired number of tests. Note that if the loop terminates due to an unsatisfiable conjunction, we might have generated less than `n` models. In this case, we can still obtain `n` test cases by combining models with different fuzzy values for remaining parameters as it was explained above.

This concludes the description of the test input generation prototype for integer preconditions. Section 8.1 details possibilities of future work on the test input generation to extend the prototype that was described in this chapter.

7 Evaluation

In this thesis, we implement a runtime assertion checker for Go programs called GoRAC. The tool generates runtime checks for specification of a program. Our goal is that GoRAC enables software engineers to detect errors in their implementations and motivate engineers to eventually verify their implementation to prove program correctness. In order to foster the use of GoRAC in Go projects, the tool has to meet the following requirements:

- The generation and execution of the runtime checks has to perform within acceptable time limits. The performance should be within appropriate bounds such that users are willing to use GoRAC and add specification to their code.
- The programmers should be able to check useful properties at runtime. Thus, the specification language supported by GoRAC needs to be expressive enough to write meaningful specification.
- The specification for runtime checking should be well-tested and simplify writing specification for a subsequent verification. Hence, the gap between assertion checking and verification should be as small as possible.

This chapter provides an extensive evaluation of the work that was done in this thesis in order to analyze whether GoRAC fulfills the above stated requirements. Section 7.1 describes the performance evaluation of the tool. Section 7.2 details the effectiveness of GoRAC's specification language, and Section 7.3 discusses the evaluation of the remaining gap between runtime checking and verification. Note that the prototype test input generation is not evaluated due to its limitations that are detailed in Chapter 6.

7.1 Performance

GoRAC's performance is evaluated for both the generation of the runtime checks with GoRAC and the execution of the runtime checks. We further discuss how in certain circumstances syntactically different but semantically equal specification can affect the performance. For all evaluations we use the Linux command `usr/bin/time` to measure the performance. Each measurement is executed multiple times; the final result is the average over all measured times. The next two subsections discuss the evaluation for the generation performance and execution performance separately. In the following, we call the generation of runtime checks based on provided specification simply the *generation*, and similarly the execution of the generated runtime checks the *execution*.

7.1.1 Generation

We measure the generation performance by generating runtime checks for the Go library Go Data Structures (GoDS) [37]. GoDS is one of the most extensive and well-maintained data structure libraries for Go containing implementations of various common data structures such as lists, heaps, stacks or trees. The library has a total of 5960 lines of code. This establishes a representative scenario for a practical use case of GoRAC. We annotate GoDS with 1347 lines of specification and generate runtime checks for all annotations with GoRAC. The total execution time of GoRAC amounts to 24 seconds. This result suggests that GoRAC performs within an acceptable time limit, thus the result validates the usability of the tool.

Limitations: Besides showing that GoRAC can be applied in practical use cases, we further analyze how syntactical changes to the specification can influence the generation time of runtime checks. Our results show that the generation times of semantically equal but syntactically different specification can diverge. This can, for instance, be observed when splitting a precondition that contains a conjunction into separate preconditions for each term of the conjunction. Figure 7.1 displays two versions of a function, one without and one with splitting the precondition, respectively. We measure the runtime check generation for two files where one contains 10 instances of the function from the top of Figure 7.1 and the other contains 10 instances of the function from the bottom. The function is duplicated

```
/*@
* requires acc(a) && acc(b) &&
*          acc(c) && acc(d) &&
*          len(a) > 0 && len(b) > 0 &&
*          len(c) > 0 &&  len(d) > 0
*/
func addFirstElements(a, b, c, d []int) int {
    return a[0] + b[0] + c[0] + d[0]
}
```

```
//@ requires acc(a)
//@ requires acc(b)
//@ requires acc(c)
//@ requires acc(d)
//@ requires len(a) > 0
//@ requires len(b) > 0
//@ requires len(c) > 0
//@ requires len(d) > 0
func addFirstElements(a, b, c, d []int) int {
    return a[0] + b[0] + c[0] + d[0]
}
```

Fig. 7.1: Two versions of a function whose specification is once a conjunction (at the top) and once multiple single terms (at the bottom)

10 times to increase the overall generation time and hence reduce the impact on the measurements by operating system noise. The file containing the specifications as conjunctions takes 16.3 seconds. The file in which the specification is split into multiple preconditions takes only 0.1 seconds.

In order to explain the large divergence between the two generation times, we profiled the GoRAC execution on the two files. The profiling showed that the main difference in the generation times stems from the parsing process of the specification expressions: Parsing multiple single terms is much faster than parsing a conjunction of all the terms. Since we did not implement the parser ourselves but instead used ANTLR [35] as a parser generator, we decided that the parsing process should be enhanced as future work. We also noted slightly longer execution times for desugaring a conjunction and extracting the old expressions from it in comparison to multiple single terms. However, the time differences for these two processes are not as profound as the one for parsing. The time differences can be explained by the overhead the recursive calls make when desugaring a conjunction. Hence, we could slightly enhance these generation times e.g. by implementing an iterative approach of desugaring and extracting old expressions.

7.1.2 Execution

The execution overhead of the runtime checks is measured by comparing the execution times of different Go files containing the same program once with and once without runtime checks. In the following, we present several programs that serve as examples for worst case scenarios of runtime overhead, i.e. they have been constructed to exhibit significant overhead when executing runtime checks. Note that for the evaluation of the execution performance, the resulting duration consists not only of the execution time but also includes compilation.

First, we measure the program given in Figure 7.2. The program computes the sum of all elements in a slice given that all elements are zero. The first column of Table

```
// If all slice entries are equal to zero, their sum will be zero
//@ requires forall i int :: i in range slice ==> slice[i] == 0
func sumZeros(slice []int) int {
    return 0
}

func main() {
    slice := make([]int, math.MaxInt32)
    sumZeros(slice)
}
```

Fig. 7.2: **Program quantifier.go:** Function with a quantifier whose runtime check leads to a significant increase in execution time

Exec. time	quantifier.go	domainorder.go		optimizations.go	
		`range a` first	`range b` first	optimized	not optimized
Plain ∅	1.12s	0.72s	0.51s	0.21s	0.25s
RACs ∅	12.4s	0.56s	14.18s	0.35s	11.75s

Tab. 7.1: Average execution times in seconds (s) of both a plain version (i.e. one without runtime checks) and a version including runtime checks for quantifier.go (Figure 7.2), domainorder.go (Figure 7.3) and optimizations.go (Figure 7.4).

7.1 shows the execution times for the program without and with runtime assertion checks. Executing the program with runtime checks, takes almost 12 times longer than the execution without the checks takes. Thus, the observation we make is that runtime checks for specifications containing quantifiers significantly increase the execution time of a program. The time difference results from the encoding of quantifiers as described in Section 3.3: The quantifier used in Figure 7.2 is translated into a for-loop that checks the condition `a[i] == 0` for all entries of the slice. Thus, executing the function with runtime checks yields a runtime of $\mathcal{O}(n)$ where n is the length of the slice whereas the function without runtime checks runs in $\mathcal{O}(1)$.

The second scenario concerns the order in which quantifier domains are stated. Figure 7.3 shows two preconditions for a function computing the intersection of disjunct slices. Both, the function at the top and at the bottom, are annotated with semantically equivalent quantifiers. The only difference is the order of domains in the quantifiers. Based on the precondition `len(a) < len(b)`, we can deduce that the first domain of the quantifier at the top ranges over the shorter slice. In contrast,

```
//@ requires len(a) < len(b)
//@ requires forall i, j int ::
//@                 _, i in range a && _, j in range b ==> i != j
func disjunctIntersection(a, b []int) []int {
    return []int{}
}
```

```
//@ requires len(a) < len(b)
//@ requires forall i, j int ::
//@                 _, j in range b && _, i in range a ==> i != j
func disjunctIntersection(a, b []int) []int {
    return []int{}
}
```

```
func main() {
    empty := make([]int, 0)
    large := make([]int, math.MaxInt32)
    disjunctIntersection(empty, large)
}
```

Fig. 7.3: **Program domainorder.go:** Two versions of a function whose preconditions have different domain orders

the first domain at the bottom ranges over the longer slice. As explained in Section 3.3, multiple domains of a quantifier result in nested for-loops. The order in which domains are stated implies the order in which they get nested: In the example on the top, the outer loop will range over the shorter slice and the inner loop over the longer one. In the example on the bottom, the nesting occurs the other way around. The main function creates two slices with the largest possible difference in number of elements to exhibit a significant impact on the execution time for the mentioned two versions of the precondition.

According to the nesting of the loops, the runtime check for the function at the top will loop over the empty domain first. Since it is empty, the quantifier immediately returns true. For the function at the bottom, the function will loop over long domain first. Thus, the execution time for the runtime checks of the function at the top is shorter than the time for the runtime checks for the function at the bottom. Our experiments confirm this explanation, as shown by the measurements in Table 7.1. We can conclude that if the lengths of domains are known, it is advisable to state shorter domains first.

A similar observation can be made when considering the order of optimized domains. Figure 7.4 shows the same example function twice but with syntactically different quantifiers. In both cases, the quantifier domains express that if the quantified variable is a member of the given slice and is bound by the interval from 0 to 100,000, then the variable is not an outlier. Following the quantifier optimizations described in Section 4.4.3, the nesting of the domains from the example at the top are optimized as follows:

```
for _, i := range values { if 0 <= i && i <= 100000 { ...  } }
```

In contrast to this, the domains for the quantifier at the bottom are not optimized and encoded as the following nested loops:

```
for i1 := 0; i1 <= 100000; i1++ { for _, i2 := range values { ...  } }
```

```
//@ requires len(values) < 100
//@ requires forall i int ::
//@          _, i in range values && 0 <= i <= 100000 ==> !isOutlier(i)
func processValues(values []int) { ... }
```

```
//@ requires len(values) < 100
//@ requires forall i int ::
//@          0 <= i <= 100000 && _, i in range values ==> !isOutlier(i)
func processValues(values []int) { ... }
```

Fig. 7.4: **Program optimizations.go:** Two versions of a precondition whose runtime check is once optimized (at the top) and once cannot be optimized (at the bottom)

The lack of optimization for the domains at the bottom of Figure 7.4 yields 100×100000 loop iterations instead of only 100 for the example at the top. This effect can also be seen in the execution times in Table 7.1. Therefore, we again recommend to check the order of quantifier domains if the runtime check overhead of a program containing quantifiers is too large.

7.2 Effectiveness of the Specification Language

We analyze the effectiveness of the GoRAC specification language regarding different properties that programmers might desire to specify. We consider a set of properties that enables thorough runtime assertion checking of programs and examine which properties can be expressed by the specification language. Furthermore, we evaluate how efficient the specification language can express these properties, i.e. how concise a property can be stated. The following paragraphs each examine different properties that are expected to be supported by a runtime checker. We first state the expectations and then determine how successful these are met by the GoRAC specification language. For the last two points, we also compare GoRAC to state-of-the-art assertion-based testing tools in order to identify differences in the employed specification language.

Reasoning about the program state: When runtime checking a program, we need to be able to reason about the current program state. The program state is defined by the contents of all memory locations, i.e. the heap and stack. With the GoRAC specification language, we can check properties of the program state by expressing conditions using valid Go expressions. E.g. we can check whether a struct field holds a certain value, whether an array is equal to an array literal or whether an expression evaluates to a desired value. Additionally, we can specify whether a function has no effects on memory using the purity annotation that GoRAC provides. However, GoRAC exhibits certain restrictions. For instance, GoRAC does currently not support type assertions. Type assertion could be included into the supported syntax as future work. Also, we always need to inquire about objects which are in scope; we are not able to directly lookup arbitrary memory addresses. When reasoning about the heap, it is also important to note that it is possible to check equality of objects but the specification language cannot express conditions about whether two objects alias.

Checking assertions at arbitrary program points: It is necessary to allow for checks at any point of the program execution such that a user can potentially ensure correctness of all visited program states for an execution. This is permitted by the different types of specification statements of GoRAC: Pre- and postconditions reason about program states before and after functions, assertion and assumption

statement can be placed at any program point inside a function, and invariants express conditions that are maintained from one loop iteration to the next. Since specification can also be part of the main function, runtime checking at the very beginning or end of a program is possible.

Comparing different program states: It is often useful to be able to relate program states at different program points with each other. Hence, a user can reason about the presence or absence of changes in between two program points. In GoRAC, one can use old expressions to refer back to previous program states. Therefore, a user can compare the current program state with old ones. Labels can be used to specify and name which program points one want to reference. This feature is also supported by other runtime assertion checkers [23, 44]. We refer the reader to Chapter 2 that discusses similarities and differences regarding old expressions of existing tools in comparison with GoRAC.

Efficient means to express specification: It is desired that a specification language provides means to write concise specification such that short annotations are sufficient to express complex properties. The specification language should make it easier for users to express properties than to implement the checks themselves. To this end, GoRACprovides predicates and quantifiers: Predicates enable reuse of runtime check annotations and therefore contribute towards writing compact specification. Quantifiers reason about all members of a domain. This permits concise specification instead of having to repeatedly specify the same condition for all members. The fact that quantifiers need to be bounded in GoRAC prohibits reasoning about infinite sets. Bounded quantifiers are a common feature of runtime checkers [23, 44]. A comparison of these tools with GoRAC concerning quantifiers is part of the related work described in Chapter 2.

7.3 Gap between Runtime Checking and Verification

As described in the motivation for this thesis in Section 1.1, runtime checking with GoRAC is supposed to supplement verification with Gobra. By enabling programmers to write specification for runtime checking a program, we hope to facilitate a later verification of the program by reusing some of the specification. In this part of the evaluation, we analyze by how far our premise that specification from runtime checking can be reused for verification holds.

The evaluation is performed in two parts: The first part consists of three case studies about the reuse of specification. The second part discusses difficulties that might be

encountered when reusing specification. In the end, a summary of the remaining gap between runtime checking with GoRAC and verification with Gobra is given.

7.3.1 Case Studies

The three case studies are based on three examples from the Viper tutorial [49]: (i) Binary search [46] is an algorithm that finds a given element in a sorted list of elements [26] (ii) Given some number k, the quickselect algorithm [48] selects the k-th largest element from a list of elements [20] (iii) Graph marking [47] is an algorithm to traverse a graph starting at some given node and mark all nodes that are reachable from the given node. Graph marking is similar to the mark-and-sweep algorithm used for garbage collection. We picked these three algorithms since they are state-of-the-art algorithms for searching, selecting and graph traversal, and we assume most readers to be familiar with them.

The Viper examples have been translated into Go and first annotated for GoRAC. In this first step, the objective was to receive GoRAC annotations that are representative of annotations an engineer without background in deductive verification would write. Therefore, only the implementations were inspired by the Viper code but the corresponding specification was created independently. In a second step, we have translated Viper specification to Gobra-like specification. Since Gobra is still under development, Gobra in its current state does not support the whole specification as it is used for this evaluation. However, we provided Gobra specification that resembles a complete specification for verification once the final Gobra tool is released. We compare the number of lines of specification (LOS) for GoRAC with the number of LOS for Gobra. Additionally, we qualitatively evaluate how much of the GoRAC specification could be reused for Gobra specification. We consider a specification annotation for GoRAC as being reusable if it semantically expresses the same condition as a specification statement at the same program point in Gobra. Table 7.2 summarizes our findings. For each case study, we shortly describe what specification was reusable. Finally, we describe the implications of our case study results with respect to the remaining gap between runtime checking and verification.

	GoRAC LOS	Gobra LOS	Reused LOS
Binary Search	16	23	6
Quickselect	28	44	10
Graph Marking	10	25	3

Tab. 7.2: Lines of specification (LOS) for GoRAC and Gobra, and reused lines of GoRAC specification for Gobra specification for three cases studies

Binary Search. Both the GoRAC and the Gobra specification require that the input slice is sorted. Both also ensure that the returned position is either equal to a slice index or equal to the value `-1` if the number that was searched for was not found. The Gobra specification includes more access permissions than the one for GoRAC . For example, the access permissions of the function are repeated as loop invariants. For some loop invariants from Gobra, the GoRAC specification expresses the condition as assertions directly in the loop body. In summary, 6 lines of the GoRAC specification were reused for the Gobra one. This entails that 37.5% (6 out of 16 LOS) of the runtime checking specification replaces approximately 26% (6 out of 23 LOS) of the specification for verification.

Quickselect. The specifications for GoRAC and Gobra have the same requirements on the input parameter `k` in the search for the k-th largest element. They also both specify properties for the elements to the left and right of the kth-largest one when the function exits. For a helper function that partitions the slice using a pivot element, the GoRAC as well as Gobra specification reasons about the position of the pivot element. For a helper function that swaps elements in the slice, both specifications use old expressions to express that the values have been swapped. There are also some differences, namely Gobra uses ghost code to specify that permutations have occured on the slice. Finally, the Gobra specification includes fold and unfold statements, and uses triggers for quantifiers. GoRAC does not support fold or unfold, and also does not need triggers in quantifiers since all GoRAC quantifiers are bounded. Overall, 10 lines of GoRAC annotations were reused for Gobra specification. This includes approximately 35% (10 out of 28 LOS) of GoRAC specification or 22% (10 out of 44 LOS) of Gobra specification.

Graph Marking. Both specifications for GoRAC and Gobra state access permissions on all nodes that make up the graph that is input to the graph marking algorithm. It is also specified by both specifications that a node cannot be marked yet when it is first visited and that it is marked after it has been visited. The propagation of markers on all adjacent nodes is included in the GoRAC as well as the Gobra specification, however, the Gobra annotation is more extensive than the GoRAC one: The Gobra specification uses ghost code to check that no nodes are being modified except for the currently considered one and all its neighbors. In this case study, GoRAC and Gobra share 3 lines of specification, i.e. approximately 33% (3 out of 10 LOS) of GoRAC specification are reused to make up 12% (3 out of 25 LOS) of the specification for Gobra.

We conclude that the part of a specification for GoRAC that reasons about the functionality of a program or function and properties of input and return parameters

can often be reused for Gobra specification. Gobra specification often extents this functional specification with annotations that are additionally needed for the correctness proof. For example, certain specifications have to be repeated at different program points to enable the automatic verifier to discharge proof obligations. Such repetitions are not present in GoRAC specification. Furthermore, we observed that certain properties have been stated as assert statements in GoRAC whereas invariants have been used in Gobra. This can be explained by the fact that in order to find bugs it is sufficient to repeatedly check an assertion. However, when proving correctness of a program, stronger guarantees such as invariants expressing properties that hold across loop iterations are needed. Gobra specification also exhibits a more extensive use of ghost code and access permissions, and also makes use of constructs like fold, unfold of quantifier triggers which are not supported by GoRAC. Finally, GoRAC does not enforce a minimal set of specification and therefore some properties might simply be forgotten in the specification. In contrast, verification is typically a repeated process of attempting a verification and extending or fixing the specification.

Ultimately, it is important to note that the findings of the three case studies are influenced by the fact that they were carried out by a person without a strong verification background. The GoRAC specification was written without having read the corresponding Viper specification that serves as a basis for the Gobra annotations. Therefore, we estimate that the case studies reflect the use of GoRAC by regular Go programmers that work on an implementation and simultaneously write specification for it.

7.3.2 Over-Approximation of Permissions

Given a struct pointer `x`, GoRAC checks an access permission for it by checking that `x` is not `nil`. Hereby, GoRAC over-approximates the access permission. I.e. any access permission that holds in a Gobra specification, also holds during runtime checking with GoRAC. However, if an access permission is deemed satisfiable by GoRAC, this does not imply that it holds during verification with Gobra. An example of such a scenario is provided in Figure 7.5. The depicted function requires access permissions

```
type foo struct {
    bar int
}

//@ requires acc(x.bar) && acc(y.bar)
//@ requires x.bar == y.bar
func exclusivity(x, y *foo) ... {
    //@ assert x == y
}
```

Fig. 7.5: Program for which executions exist that pass the runtime checks generated by GoRAC while verification with Gobra fails

for the field `bar` of the two parameters `x` and `y` that are pointer to a struct of type `foo`. In the body of the function, we assert that the two parameters are equal. Runtime checks generated by GoRAC validate that both `x` and `y` are not `nil`, that the structs have equivalent values for the field `bar`, and the runtime check for the assertion finally checks whether the two struct pointers are equal. If both parameters receive the same input value, the runtime checks succeeds. Thus, there exists an execution that satisfies the specification. However, in the case of verification with Gobra, the program does not successfully verify: Access permissions granted for a location are exclusive. Hence, it is not possible to hold multiple permissions to the same location [49]. In other words, write access prohibits aliasing. In the example below, the access permissions `acc(x.f)` and `acc(y.f)` thus imply that `x` and `y` do not alias. Consequently, it is guaranteed that `x` is not equal to `y`.

The over-approximation of permissions can thus yield controversial outcomes of runtime checking and verification. This also contributes to the remaining gap between runtime checking and verification.

This concludes last part of the evaluation for this thesis. We have shown that GoRAC performs within acceptable time limits both for generation and execution of the runtime checks. We have discussed circumstances in which the performance is influenced by the way specification is stated. The evaluation has further shown that the effectiveness of the GoRAC specification language covers most of the expectations for runtime checking tools with only some minor limitations. Finally, the evaluation suggests that runtime checking with GoRAC is a useful supplement to verification with Gobra despite the differences that still exist.

8 Conclusion

This thesis detailed the design and implementation of GoRAC, a runtime checking framework based on assertions for the Go programming language. We provided an extensive description of GoRAC's specification language which we designed to be close to the specification language of the Go verifier Gobra. This allows for reuse from runtime checking specification for verification. It is defined how runtime assertion checks are generated for the various annotations that can be expressed in the specification language. In combination with the generated checks, a technique for test input generation is presented which can be used to test a program against its specification during execution. Thus, this thesis promotes the use of specification through the benefit of automatic testing.

We evaluated the work in this thesis with regard to the performance of generating and executing runtime checks. The evaluation showed that runtime check generation performs within an acceptable time limit, hence validating the usability of the GoRAC tool. The thesis further discussed how syntactical changes to the specification can influence the runtime of checks. Moreover, the evaluation suggested that GoRAC is a helpful supplement to Gobra since we could adopt various GoRAC annotations when writing Gobra specification in different case studies. Therefore, this thesis contributes towards bridging the gap between runtime checking and verification.

8.1 Future Work

There exist three main opportunities for future work on GoRAC. First, GoRAC should add complete support of exclusive old expressions as described in Section 4.6. The exclusive old algorithm outlined in Subsection 4.6.5 can be used as a reference. The thesis further discusses some implementation specifics for exclusive old expressions in Section 4.6.7. These concepts need to be generalized for all types of expressions but can serve as a first basis for the implementation of the exclusive old expression support in GoRAC.

Second, the prototype for the test input generation detailed in Section 6 should be extended to handle all types of parameters. Besides the already existing support for integer parameters, this includes handling all other primitive types and complex data

structures of Go like arrays or slices. Subsection 6.2 explains how Gobra can be used to support complex Go data structures. Thus the SMT solver Z3, which is currently part of the test input generation prototype, should be replaced with a dependency on Gobra. Additionally, we can expand the test input generation by considering certain edge cases, e.g. positive or negative integer values, empty data structures, or null pointers. Once a test input generation for arbitrary parameter types has been implemented, it needs to be evaluated how efficiently errors are detected by the generated tests.

Finally, more features of Gobra's specification language can be added to the specification language of GoRAC. Permission tracking as described in Section 3.5 can be augmented with support for `inhale` or `exhale` statements. These statements are used in Gobra to add or remove permissions. Furthermore, predicate support in GoRAC can be enhanced by handling `fold` and `unfold` operations. An `unfold` operation exchanges a predicate instance for its body, whereas a `fold` operation exchanges a predicate body for a predicate instance [49]. Finally, ghost code can be incorporated into GoRAC's specification language to provide more means for auxiliary declarations.

Bibliography

[1] Junade Ali. Go by Contract. www.github.com/IcyApril/gobycontract. [Online; accessed 2020-05-08].

[2] Linard Arquint, Martin Clochard, Peter Müller, Wytse Oortwijn, and Felix Wolf. Gobra. www.pm.inf.ethz.ch/research/gobra.html. [Online; accessed 2020-05-14].

[3] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[4] Michael Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Communications of the ACM*, 54:81–91, 06 2011.

[5] Armin Biere, Marijn Heule, Hans von Maaren, and Toby Walsh. Handbook of Satisfiability. *Frontiers in Artificial Intelligence and Applications*, 185, 2009.

[6] Sergey Bronnikov. go-contracts. www.github.com/ligurio/go-contracts. [Online; accessed 2020-05-08].

[7] Maria Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, 2015.

[8] Lori Clarke and David Rosenblum. A Historical Perspective on Runtime Assertion Checking in Software Development. *ACM SIGSOFT Software Engineering Notes*, 31:25–37, 05 2006.

[9] Beman Dawes, David Abrahams, and Rene Rivera. Boost C++ Libraries. https://www.boost.org/. [Online; accessed 2020-10-12].

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[12] Carlo Furia, Bertrand Meyer, and Sergey Velder. Loop Invariants: Analysis, Classification, and Examples. *ACM Computing Surveys*, 46, 11 2012.

[13] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Code Contracts. http://research.microsoft.com/contracts. [Online; accessed 2020-10-12].

[14] The Go Authors (golang.org/AUTHORS). Go Language Specification. https://golang.org/ref/spec. [Online; accessed 2020-09-15].

[15] The Go Authors (golang.org/AUTHORS). Go Package `ast`. www.golang.org/pkg/go/ast. [Online; accessed 2020-05-18].

[16] The Go Authors (golang.org/AUTHORS). Go Package `parser`. www.golang.org/pkg/go/parser. [Online; accessed 2020-05-14].

[17] The Go Authors (golang.org/AUTHORS). Go Package `types`. www.golang.org/pkg/go/types. [Online; accessed 2020-05-14].

[18] The Go Authors (golang.org/AUTHORS). Go: The programing language. www.golang.org/. [Online; accessed 2020-05-07].

[19] Robert Hierons, Kirill Bogdanov, Jonathan Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony Simons, Sergiy Vilkomir, Martin Woodward, and Hussein Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41:1–76, 02 2009.

[20] Charles Anthony Richard Hoare. Algorithm 65: Find. *Communications of the ACM*, 7:321–322, 1961.

[21] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *Lecture Notes in Computer Science*, 6617:41–55, 04 2011.

[22] Alexander Kunze. *Efficient Automated Testing Framework for the SCION Architecture*. Bachelor Thesis, Network Security Group, Department of Computer Science. Eidgenoessische Technische Hochschule Zuerich, 2019.

[23] Gary Leavens, Albert Baker, and Clyde Ruby. JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, 12:175–188, 1999.

[24] Gary Leavens and Yoonsik Cheon. Design by Contract with JML. 2004.

[25] Insup Lee, Hanêne Ben-Abdallah, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. A Monitoring and Checking Framework for Run-time Correctness Assurance. *Departmental Papers (CIS)*, 12 1998.

[26] Derrick Lehmer. Teaching Combinatorial Tricks to a Computer. *Proceedings of Symposia in Applied Mathematics*, 10:180–181, 1960.

[27] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. *Lecture Notes in Computer Science*, 6355:348–370, 12 2010.

[28] Eva Charlotte Mayer. GoRAC - Go Runtime Assertion Checker. https://gitlab.inf.ethz.ch/OU-PMUELLER/student-projects/go-runtime-checking. [Online; accessed 2020-10-29].

[29] Bertrand Meyer. *Object-Oriented Software Construction (2nd Ed.)*. Prentice-Hall, Inc., USA, 1997.

[30] Sun Microsystems. Javadoc - The Java API Documentation Generator. https://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/javadoc.html. [Online; accessed 2020-10-23].

[31] Peter Müller and Joseph Ruskiewicz. Using Debuggers to Understand Failed Verification Attempts. In *Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 73–87, 06 2011.

[32] Peter Müller, Malte Schwerhoff, and Alexander Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer-Verlag, 2016.

[33] Peter Müller and Ina Schäfer. *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*. Springer, 2018.

[34] Martin Odersky. Contracts for Scala. In Barringer H. et al., editor, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 51–57, 11 2010.

[35] Terence Parr. ANTLR. https://www.antlr.org/. [Online; accessed 2020-10-02].

[36] Terence Parr. The Definitive ANTLR 4 Reference. *Pragmatic Bookshelf*, 1:328, 01 2013.

[37] Emir Pasic. GoDS (Go Data Structures). https://github.com/emirpasic/gods. [Online; accessed 2020-10-29].

[38] Ian C. Pyle. *ADA Programming Language*. Prentice Hall PTR, USA, 1981.

[39] Marco Ristin. gocontracts. www.godoc.org/github.com/Parquery/gocontracts/gocontracts. [Online; accessed 2020-05-08].

[40] Daniel Smith and Tim Hockin. gofuzz. www.github.com/google/gofuzz. [Online; accessed 2020-05-18].

[41] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.

[42] Ruslan Stepanenko. Design-By-Contract for Go. www.godoc.org/github.com/drblez/dbc. [Online; accessed 2020-05-08].

[43] Dmitry Vyukov. go-fuzz: Randomized Testing for Go. https://github.com/dvyukov/go-fuzz. [Online; accessed 2020-10-16].

[44] Karen Zee, Viktor Kuncak, and Martin Rinard. Runtime Checking for Program Verification Systems. *Lecture Notes in Computer Science*, 4839:202–213, 01 2007.

[45] Daniel Zimmerman and Rinkesh Nagmoti. JMLUnit: The Next Generation. 6528:183–197, 06 2010.

[46] Programming Methodology Group ETH Zurich. Binary Search Example from the Viper Tutorial. http://viper.ethz.ch/examples/binary-search-array.html. [Online; accessed 2020-09-30].

[47] Programming Methodology Group ETH Zurich. Graph Marking Example from the Viper Tutorial. http://viper.ethz.ch/examples/graph-marking.html. [Online; accessed 2020-09-30].

[48] Programming Methodology Group ETH Zurich. Quickselect Example from the Viper Tutorial. http://viper.ethz.ch/examples/arrays_quickselect_rec.html. [Online; accessed 2020-09-30].

[49] Programming Methodology Group ETH Zurich. Viper Tutorial. http://viper.ethz.ch/tutorial/. [Online; accessed 2020-09-30].

List of Figures

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.