3 Specification Language

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 $\mathtt{x}$, 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 `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 $m(\texttt{L}) = \hbar^L$. Then, we define the following old semantics for variables:*

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

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:*

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

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:*

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

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

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:

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

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.

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.