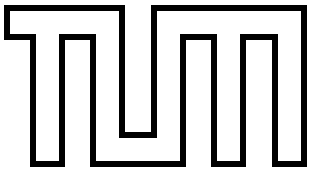


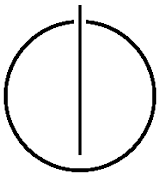
FAKULTÄT FÜR INFORMATIK

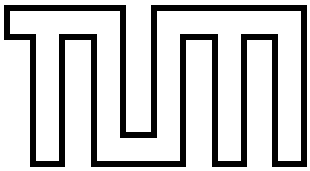
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Quantified Permissions for Sets and Sequences

Korbinian Breu





# FAKULTÄT FÜR INFORMATIK

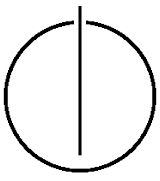
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

## Quantified Permissions for Sets and Sequences

## Quantifizierte Zugriffsrechte für Mengen und Sequenzen


| | |
|---|---|
| Author: | Korbinian Breu |
| Supervisors: | Prof. Dr. Tobias Nipkow (TUM) |
| | Prof. Dr. Peter Müller (ETH Zürich) |
| Advisors: | Dr. Ioannis Kassios (ETH Zürich) |
| | Dr. Alexander Summers (ETH Zürich) |
| Date: | March 1, 2014 |

I assure the single handed composition of this Master's thesis only supported by declared resources.

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, on March 1, 2014 Korbinian Breu

# Abstract

Consider a parallel mergesort: it splits an array into two parts, sorts each part in a separate thread, and then merges the results. We want to implement this algorithm correctly, and we want the correctness of the implementation to be automatically verified. More generally, we want to verify the correctness of algorithms that operate on unbounded data. When implementing mergesort, for example, the size of the array is not statically known.

Silicon is an automatic verifier for concurrent programs, based on symbolic execution. Before this project, one had to specify algorithms on unbounded data structures with recursive abstract predicates. These predicates dictate a specific traversal of the data structure and thus do not harmonise with algorithms that traverse in a different way. For example, binary search is hard to specify on an array that is modelled as a linked list. Thus, the specification of such algorithms with abstract predicates is unnatural and does not resemble the implementation.

In this thesis, we describe how we extended Silicon to support permissions under universal quantifiers. Permissions model a thread's access to heap cells, and we use permissions under universal quantifiers to model the access to elements of a set or a sequence. We show how we use permissions under quantifiers to specify permission-related aspects of algorithms on unbounded data structures. Unlike predicates, our approach does not dictate a specific traversal.

With our new feature, Silicon verifies the permission-related specification of algorithms on unbounded data. Among others, we verified mergesort, a thread-safe mutable array, and a union-find structure.

# Acknowledgments

First of all, I want to thank my advisors, Ioannis Kassios and Alex Summers. You not only shaped this thesis, but I also learned a lot from you. Alex, thank you for the clarity of your explanations. You taught me to try to understand problems before coming up with (possibly premature) solutions. Ioannis, thank you for specifying examples, testing my implementation, and pointing out errors and incompletenesses. Your continuous feedback was essential. I look forward to our future work.

Prof. John Boyland and Malte Schwerhoff, I want to express my gratitude for your great input. John, thank you for taking part in our meetings. Your ideas and feedback directly influenced this thesis. Malte, you do an amazing job at ensuring the code quality of Silicon. The preciseness of the rules and functions presented in this thesis would be different without your rigorous focus on good code.

Prof. Peter Müller, thank you for giving me the opportunity to write this thesis at your chair. I did not expect it to be so uncomplicated to write a thesis in an exchange semester. Prof. Tobias Nipkow, thank you for being the official supervisor at TUM. Without your challenging Semantics lecture, I might not have chosen such a thesis topic.

Thomas, thank you for organising our social life in Zurich. Andreas, Eli, Isabella, Kristina, Matthias, Marcel, Michaela, Sebastian, Stefan, Tanja, and Wolf, thank you for making the time outside my thesis so amazing. You really excelled in making my stay here in Zurich more than awesome.

Mom and dad, Magdalena and Felizitas, thank you for supporting me. I promise to come home more often.

Annika, this thesis is dedicated to you.

# Contents

# 1 Overview

## 1.1 Motivation

Have you ever implemented thread-safe collections? When we develop parallel algorithms, we often make mistakes.

Consider a thread-safe mutable set: many threads can read the elements. But, to prevent indeterminacy in the result, a thread that modifies elements requires exclusive access to these elements. Or, consider a parallel mergesort: it splits an array into two parts, sorts each part in a separate thread, and then merges the result. Both are algorithms that operate on unbounded data: for mergesort, e.g., the size of the array is not statically known.

We want to implement these algorithms correctly, and we want the correctness of the implementation to be automatically verified. Therefore, we have to prove that the implementation adheres to a specification of the algorithm's properties. Mergesort, for example, should never access a part of the array which is being sorted by another thread.

Many automated verifiers for concurrent programs, e.g. Verifast [JP08], Chalice [LMS09], jStar [DP08], VeriCool [SJP08], and Silicon [Sch11] use the concept of permissions to control the access to heap structures: at each program point, a thread has either read, write, or no access to a given heap location.

When the number of accessed locations is statically unbounded, it is impossible for a specification to explicitly describe each of these accessed locations [SD13]. In the mentioned verifiers, the only option to specify permissions for a statically unknown number of locations were *recursive abstract predicates*. A predicate is given as part of the program's specification and has an assertion as its body. The body may contain instances of the same predicate, making the predicate recursive. With such a recursive predicate, one can implicitly require permission to access, e.g., every `next` and `value` field in a linked list. The disadvantage of specifying permissions in this way is that the recursive nature of a predicate dictates a specific traversal of the data structure. Consider Figure 1.1, which shows the typical heap structure of a linked list specified by a recursive predicate. It is designed to be traversed from its beginning to its end. This implicitly dictated traversal complicates correctness proofs of algorithms that traverse in a different way.

For example, imagine your model of an array looks similar to the specification of a linked list. You decided to model it this way because your algorithms traverse the array from start to end. Now, imagine you would like to implement binary search. The implementation of binary search compares the `value` field with the `value` field of the center element, then with the `value` field of the center element of the upper or lower half, and proceeds to divide the remaining sequence. Intuitively, it would be hard to show that permission to

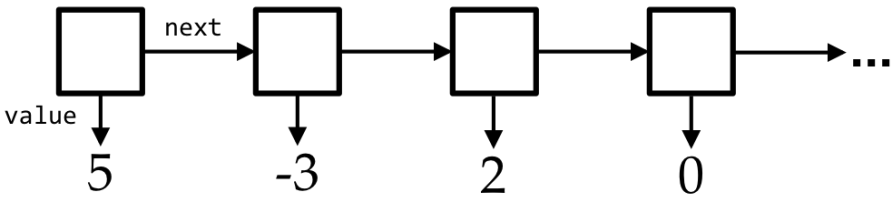



Figure 1.1: Recursive predicates implicitly dictate a specific traversal.

access the `value` field of the center element is held, as it is unknown how often the body of the predicate has to be "unrolled". The verifier should offer a mechanism to specify the permissions to access such data structures *without* dictating a specific traversal. We propose such a new mechanism.

This mechanism allows to specify access permissions for *sets* and *sequences* without dictating a specific traversal. Sets are unordered collections of mutually distinct elements, sequences are ordered collection of elements that do not need to be distinct. We show how to specify the mentioned algorithms and data structures with our new mechanism. Our proposal extends the specification language with permissions under universal quantifiers. It is implemented as an extension of the verification methodology used in Silicon, a verifier for the intermediate verification language SIL.

To our best knowledge, no automated tool based on symbolic execution supports permission under quantifiers.

## 1.2 Silicon and its dependencies

- **SIL** is an intermediate verification language based on the concepts of implicit dynamic frames (IDF) [SJP09] and fractional permissions [Boy03]. It has sets and sequences as generic collection types.
- **Silicon** is a verifier for SIL based on symbolic execution. The project described in this report extends Silicon to support assertions that contain permissions under universal quantifiers. Such assertions were allowed syntactically in SIL before, but not supported by any verifier.
- **Z3** is a SMT solver [DMB08]. It handles proof obligations that arise during the symbolic execution of programs. This component is also called the *prover*.

## 1.3 Permissions under quantifiers

Specifications in SIL, i.e. method preconditions and postconditions, and loop invariants, are written as assertions in the logic of IDF. This logic is similar to first-order logic, but adds constructs to specify the access to heap locations. It already syntactically allows for

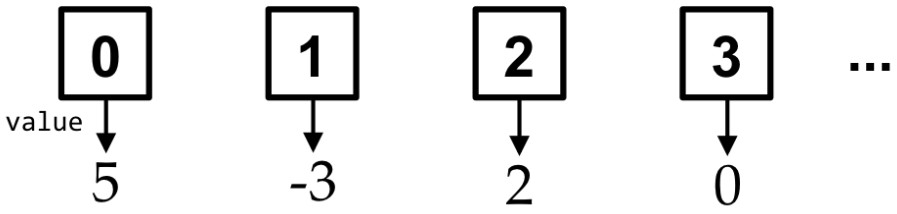



Figure 1.2: Permissions under quantifiers allow to specify random access for a sequence.

a universal quantifier. We added support for permissions under universal quantifiers to express assertions such as "at this program point, it is known that read access to the field `value` for all objects in set $S$ is held", or "for each index $i$ of valid indices for sequence $S$, access to the field `value` of the object at $S[i]$ is held". When we model an array with a sequence, this resembles Figure 1.2. We omit the `next` pointer needed above, and instead access elements by their index.

Thus, in contrast to recursive predicates, permissions under quantifiers allow to specify access without fixing how this access is used. For example, they allow any traversal of a sequence.

## 1.4 Implementation outline

Silicon keeps a representation of the content of the heap in the verifier, separate from the prover. Whenever it needs to prove an assertion about heap contents, Silicon first uses Z3 to prove if the program may access the heap content in the current state. If the program holds the required permissions, Silicon encodes the assertion with the current heap content and lets Z3 prove it. Silicon passes only formulas to the prover.

To support sets and sequences, we modified the representation of the heap. As soon as Silicon encounters permissions under quantifiers for a certain field, it changes the structure of the heap for this field to a new structure. Consequently, Silicon also modifies the operations on contents of the heap, i.e. permission lookup, permission transfer, and keeping track of the value of heap cells. It then transforms permissions under quantifiers to fit this representation of the heap. To fulfil new proof obligations arising from that new heap structure, we added new functions in the prover and axiomatised them.

Our approach is extensible such that further projects can implement new assertions that are not yet handled. We separated permissions under quantifiers into so-called *idioms*. The two available idioms are sets and "splitting sequences". "Splitting sequences" is an idiom that can handle assertions that are constrained to a range of indices of a sequence. We built it to support e.g. how mergesort splits an array in two parts.

Silicon's old heap representation is still important. It generates fewer calls to Z3, and is thus significantly faster. So, Silicon keeps the fields without permissions under quantifiers in the old representation.

## 1.5 Results

We verified the permission-related part of interesting algorithms that can be specified with sets and sequences. These include mergesort, binary search, a thread-safe mutable array, and a union-find structure. We verified many more artificial programs that now serve as a test suite. Our extension does not affect the performance of Silicon for programs without permissions under quantifiers substantially.

# 2 Background

## 2.1 The Semper Intermediate Language SIL

The Semper Intermediate Language SIL is an imperative language targeted at static verification. It uses assertions such as pre- and postconditions as well as loop invariants to specify the expected program behaviour. It is called an *intermediate* language because programmers rarely interact directly with SIL; rather, they use a front-end tool to translate from their source language to SIL[1]. Then, they use a back-end tool such as Silicon to attempt to verify the program.

### 2.1.1 Language concepts

SIL does not explicitly encode classes. All fields are globally declared and available to all objects. All objects are referred to by a basic reference type, `Ref`.

At each point of a SIL program, it is known which fields of which objects may be read or written. The idea is to *frame* methods: to compute an upper bound on the locations that are read or written. Framing methods ensures modularity: each method implementation can be verified on its own, without considering the implementation of other methods [SJP09].

In SIL assertions, access to a field is expressed as `acc(e.f,p)`, where `e` evaluates to an object that is called the receiver, `f` is a field name, and `p` is a permission amount with $0 < \texttt{p} \leq 1$. Having permission $1$ to a field means write access (we also use `write` as $1$). Having permission $0 < \texttt{p} < 1$ means read access, and permission $0$ means no access. Write access is exclusive, and the sum of all permissions to a field at any given time is $\leq 1$. This model of specifying access rights is called *fractional permissions* [Boy03].

An assertion may conjoin more than one such access predicate, e.g. the assertion `acc(x.f, 1/2) && acc(x.f, 1/2)`. In SIL, `&&` is not the logical conjunction ($\wedge$), but the so-called *separating conjunction*[2]. Its semantics allows us to "add up" permissions that are given to the same receiver. So, if the earlier assertion holds, we actually have full access to `x.f`. If either side of the separating conjunction is pure, i.e. it does not contain access predicates, then its semantics degenerates to the usual logical conjunction. This specification language is a variant of IDF [SJP09].

Permissions can be transferred, e.g. between two threads or two method executions of the same thread. For example, if a method happens to call another method, it must give away

[1] Examples for frontends are Scala2SIL and Chalice2SIL.
[2] Classically, in separation logic, the separating conjunction is written as the star $*$.

the permissions required by the method, and in return it gains the permissions ensured by the method. Giving away permissions, e.g. via the postcondition or before a method call, is called *exhaling*. Roughly speaking, exhaling an assertion means proving that an assertion holds, and in particular the permissions required are held. Gaining permissions, e.g. via the precondition or after a method call, is called *inhaling*. Inhaling an assertion assumes it, and obtains the permissions in the assertion for the current thread. If permissions to a location `o.f` are inhaled to which no permission was previously held, an arbitrary value is assigned. This models the fact that another thread might have modified the location since the current thread last accessed it [LM09]. Expressing permission as fractions makes sense: a method can distribute permissions among its callees, i.e. letting each callee read some field, and can then collect and still provably have its full permission back.

SIL has no explicit notions of concurrency, e.g. threads or monitors. Instead, these concepts can be encoded in SIL with fractional permissions and permission transfer[3]. Hence, there is no need to support them explicitly. Intuitively, write permission is exclusive, no other thread can read the location. Read permission can be shared among multiple threads.

There are cases where the number of accessed fields is not statically known. For recursive data structures, such as a linked list, SIL provides *recursive abstract predicates*. A predicate is an assertion (called its *body*) with a name. The *fold* and *unfold* operations exchange the predicate name for its body (and vice versa). Permissions are also held to access predicate instances. So, more precisely, unfold exhales permission to a predicate instance and inhales the predicate's body, and fold exhales the predicate's body and inhales permission to a new predicate instance. If a predicate is recursive, its body contains an instance of the same predicate. Recursive predicates can be used to model recursive data structures.

Listing 2.1: A linked list can be specified with recursive abstract predicates.

```
1  var value: Int
2  var next: Ref
3
4  predicate list(this:Ref) {
5       acc(this.value, write) &&
6       acc(this.next, write)  &&
7       (this.next != null ==> acc(valid(this.next), write))
8  }
9
10 method add(this:Ref, value:Int)
11 requires acc(valid(this), write)
12 ensures acc(valid(this), write)
13 {
14     unfold acc(valid(this), write)
15     if(this.next != null) {
16          add(this.next, value)
17     } else {
18        var node:Ref
19        node := new()
20        node.next := null
21        node.value := value
22        fold acc(valid(node), write)
```

[3] Chalice2SIL encodes Chalice concurrency constructs in SIL.

```
23          this.next := node
24      }
25      fold acc(valid(this), write)
26  }
```

In Listing 2.1, the permissions for a linked list are specified in the recursive predicate `list`. This predicate contains permission `acc(this.value, write)` to access a node's value, and `acc(this.next, write)` to its pointer to the next element. In case the given node is not the last one, it also ensures the access to the rest of the list. Method `add` recursively unfolds this predicate until it reaches the end of the list, and then adds a new node.

### 2.1.2 Set and sequence syntax

SIL supports sets and sequences, and common operations on them. A set is a collection of distinct objects, while a sequence is an ordered and indexed collection of elements that are not necessarily distinct. Both sets and sequences are value types. The supported syntax is illustrated in Listing 2.2.

Listing 2.2: SIL supports common operations for sets and sequences

```
1  method sets() {
2      // Set(x,y,...) constructs an explicit set
3      var a:Set[Int] := Set(1,2,3)
4      assert 2 in a
5      assert Set(1) subset a
6      assert 3 == |a| // |S| denotes the size of a set
7      assert Set(1,2,3,4) = a union Set(4)
8      assert Set(1) = a intersection Set(1)
9      assert Set(1,2) = a setminus Set(3)
10 }
11 method sequences() {
12     // Seq(x,y,...) constructs an explicit sequence
13     var c:Seq[Int] := Seq(1,2,3,4,5)
14     assert c[0] == 1
15     // S[i:=j] is S updated at index i to value j
16     var b:Seq[Int] := c[0:=-1]
17     assert b[0] == -1
18     // S[i..] drops all indices less than i
19     assert c[1..] == Seq(2,3,4,5)
20     // S[..i] takes all indices less than i
21     assert c[..1] == Seq(1)
22     // S[i..j] is the subsequence from i inclusive to j exclusive
23     assert c[1..3] == Seq(2,3)
24     // S1 ++ S2 concatenates two sequences
25     assert Seq(1,2,3) == Seq(1,2) ++ Seq(3)
26     // |S| denotes the length of a sequence
27     assert |c| == 5
28     // [i..j] denotes the sequence of integers from i inclusive to j
           exclusive
29     assert [1..3] == Seq(1,2)
30 }
```

## 2.2 Silicon

Silicon is a back-end for SIL. It takes a SIL program, and is either able to verify its validity, or outputs potential bugs. Silicon is based on a verification methodology called *symbolic execution*. It is sound and incomplete.

Intuitively, symbolic execution means executing the program with fixed, but unknown (symbolic) values. Each assertion and statement modifies the symbolic state, which tracks the current local variables, contents of the heap and permissions to access these contents, and assumptions gained about symbolic values. How each program construct modifies the state is formalized in symbolic execution rules. Proof obligations that arise, e.g. when proving that a postcondition holds, are fulfilled by the prover, which, in Silicon's case, is the SMT solver Z3 [DMB08].

We are mostly interested in the model of the symbolic heap, as supporting permissions under quantifiers means modifying the way Silicon keeps track of the heap contents and permissions. Silicon represents the state of the symbolic heap with so-called *heap chunks*. Heap chunks represent currently accessible memory cells, the permission which is held to them, and their symbolic values. In particular, we consider *field chunks*, which hold the permission and the value of a single field location. Later, we will extend the heap model.

In sections 2.2.1 to 2.2.3, we present an extended version of the formalism by Schwerhoff [Sch11]. In his work, Schwerhoff describes the formalism and implementation of Silicon. Schwerhoff's formalism itself builds on the work by Smans et al. [SJP10].

### 2.2.1 Formalism

**Definition 1** (Language)**.** The following sets are used to refer to SIL code in definitions of our symbolic execution algorithm:

- $\mathcal{X}$, the set of variables with typical element $\mathtt{x}$
- $\mathcal{E}$, the set of expressions with typical element $\mathtt{e}$
- $\Phi$, the set of assertions with typical element $\phi$
- $\mathcal{S}$, the set of statements with typical element $\mathtt{s}$
- $\mathcal{F}$, the set of field names with typical element $\mathtt{f}$

⌟

Assumptions gained during the symbolic execution of a program, e.g. from the precondition of a method, are encoded as first-order logic terms and formulas.

**Definition 2** (Term)**.** We use the word *term* for any first-order term or formula. The set of terms is called $T$, with typical element $t$. ⌟

Intuitively, each permission $\mathtt{e} \in \mathcal{E}$ can be symbolically evaluated to a term, as long as enough permissions are held to access needed heap contents. We use $t_\mathtt{e}$ to denote the term

that results in the evaluation of `e`, assuming that the evaluation succeeded. So, examples of terms are $t_{\mathtt{x}} = t_{\mathtt{y}}$ encoding the expression `x==y`, and $t_{\mathtt{x}} \in t_{\mathtt{S}}$ encoding the expression `x in S`.

**Definition 3** (Sort)**.** Each term has a corresponding sort. We are mainly interested in the sorts *Int*, *Bool*, *Ref*, *Perm*, and the generic sorts *Set* and *Seq*, where *Int* and *Bool* are integers and booleans, respectively, *Ref* are object references, *Perm* are fractional permissions, *Set* are sets and *Seq* are sequences with some sort as element type. We define the following subsets of $T$:

- $T_R \subset T$, the set of reference terms with typical element $t_r$
- $T_\alpha \subset T$, the set of fractional permission terms with typical element $t_\alpha$
- $T_{Bool} \subset T$, the set of boolean terms with typical element $t_{bool}$

⌟

**Definition 4** (Sort mapping)**.** All of the types in SIL can straightforwardly be mapped to a sort. We write $sort(\mathtt{f})$ to get the sort of the type of a field name `f`. ⌟

**Definition 5** (Symbolic state)**.** A symbolic state is a 4-tuple $(\gamma, h, g, \pi) \in \Sigma$ composed of a symbolic store $\gamma$, a symbolic heap $h$, a symbolic old heap $g$ and path conditions $\pi$, all of which will be defined in the rest of this subsection. ⌟

The symbolic state contains all information available while symbolically executing the program. It determines the result of assertion and expression evaluations, and is modified by assertions and statements. Access to a component $c \in \{\gamma, h, g, \pi\}$ of a state $\sigma$ is denoted by dot-syntax: $\sigma.\gamma$, $\sigma.h$, $\sigma.g$, and $\sigma.\pi$. Substituting a concrete state component $c'$ for a component $c$ yields a new state $\sigma'$ and is denoted by $\sigma[c := c']$. For example, $\sigma[h := \emptyset]$ clears the current heap.

**Definition 6** (Symbolic store)**.** A symbolic store $\gamma \in \Gamma$ is a partial function $\gamma : \mathcal{X} \to T$ from variables to terms. Its domain comprises all variables in the current scope of the symbolic execution. ⌟

We follow [Sch11] and introduce the shorthand $\sigma + (\mathtt{x}, t)$ to add a variable to a state, and add new notation $\sigma - \mathtt{x}$ to remove a variable from the state[4]. Adding a variable corresponds to pushing it into the current scope, and removing means that a variable is popped out of scope, e.g. after evaluating a quantifier.

**Definition 7** (Field chunk and field terminology)**.** A *field chunk* is of the form $t_r.\mathtt{f} \mapsto t_v \# t_\alpha$, where

- $t_r \in T_R$ represents a *receiver object*
- $\mathtt{f} \in \mathcal{F}$ represents a *field name*
- $t_v \in T$ represents the *field value*
- $t_\alpha \in T_\alpha$ is a fractional permission representing the access to the *field location* $t_r.\mathtt{f}$.

[4] Strictly speaking, these are function updates of the component $\sigma.\gamma$. [Sch11] contains a more rigorous introduction.

If such a field chunk is part of the heap, then (1) the field location $t_r.\mathtt{f}$ may be accessed with permission $t_\alpha$ and (2) the location's current value is $t_v$.

Let *CH* be the set of all field chunks. ⌟

**Definition 8** (Symbolic heap)**.** A symbolic heap $h \in H$ is a set of field chunks representing currently accessible memory cells and their values[5].

A chunk $c \in CH$ can be added to and removed from a heap $h$, which is denoted by $h' = h+c$ and $h' = h - c$, respectively. ⌟

**Definition 9** (Path conditions)**.** A set $\pi \in \Pi$ of *path conditions* is a set of first-order logic formulas representing assumptions gained from the symbolic execution in progress. ⌟

For example, for a given field access `x.f` and a local variable `y` with current values $t_{\mathtt{x.f}}$ and $t_{\mathtt{y}}$, respectively, an assumption `x.f > y` would go to the path conditions as $t_{\mathtt{x.f}} > t_{\mathtt{y}}$. Sources of path conditions are preconditions, postconditions of called methods and guards of if-then-else statements. Path conditions can also arise from the evaluation of expressions and inhale statements.

We update path conditions $\pi$ by adding a term $t$, denoted by $\pi' = \pi + t$.

In symbolic execution, we need a way to assign unused symbolic terms to variables. For example, when verifying a method, all the method parameters need to be assigned such values. We call these values *fresh*.

**Definition 10** (Fresh)**.** An atomic term (a variable) or a function is called fresh with respect to a set of states if it is syntactically distinct from all terms and functions, respectively, in these states. We write $t = \mathit{fresh}_S$ to express that $t$ is such a term of sort $S$, or $t_1 = \mathit{fresh}_{S_1 \to S_2}(t_2)$ to express that $t$ is a function application term of a fresh function from $S_1$ to $S_2$ applied at a term $t_2$ of sort $S_1$. Note that $\mathit{fresh}_{S_1 \to S_2}$ itself is not a term, as only first-order logic terms are allowed. ⌟

The explicit naming of the sort is a hint for the reader, and to make it easier to distinguish between fresh atomic terms and functions.

We may use *fresh* in places where a term is expected, e.g. $t_r.\mathtt{f} \mapsto \mathit{fresh}_{sort(\mathtt{f})} \# t_\alpha$.

The correctness of the rules of our symbolic execution algorithm depends on the freshness property.

### 2.2.2 Symbolic execution rules

In the symbolic execution rules we use $\sigma \vdash_{\mathtt{z3}} \phi$ to denote that a formula $\phi$ has to be proved in state $\sigma$. The $_{\mathtt{z3}}$ subscript is there to emphasise that these proof obligations are handled by Z3.

The rules are presented in *continuation-passing style* (CPS) [FWH01]. In CPS, each function has, as an extra argument, a function representing the remainder of the computation. For

[5] Other types of chunks exist which are orthogonal to this work.

example, $f(\ldots, g)$ computes the result of $f$, then invokes the continuation $g$ with this result, thereby continuing with the remaining computation. In the definition of the following rules, the last (function) argument is the continuation.

Expressions are evaluated by the function **eval**:

- **eval** : $\Sigma \times \mathcal{E} \times (\Sigma \times T \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$, evaluates an expression into a term. It is implemented with the helper function
- **eval'** : $\Sigma \times \Pi \times \mathcal{E} \times (\Pi \times T \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$[6], where the second argument accumulates path conditions yielded by the ongoing evaluation.

*Producing* assertions corresponds to the concept of inhaling. In its simplest form, conjuncts of the assertion that are access predicates `acc(e.f,p)` are evaluated and a chunk $t_{\mathsf{e}}.\mathtt{f} \mapsto \mathit{fresh}_{sort(\mathtt{f})} \# t_{\mathsf{p}}$ is put on the heap. Pure parts of the assertion is evaluated and the resulting term is used as an additional assumption during the rest of the execution. Producing an assertion can affect the current heap and the path conditions.

Assertions are produced by the function

- **produce** : $\Sigma \times T \times T_\alpha \times \Phi \times (\Sigma \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$, where the second argument is the *snapshot* term used to produce the assertion[7], and where the third argument is a factor used to scale access permissions that are to be produced[8]. **produce** is defined with the helper function
- **produce'** : $\Sigma \times T \times T_\alpha \times \Phi \times (H \times \Pi \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$, which passes only the newly produced heap chunks and path conditions to the continuation.

*Consuming* assertions corresponds to the concept of exhaling. For parts of the assertion that are pure, the consumption attempts to evaluate them to a term and then prove them in the current state. For conjuncts of the assertion that are access predicates `acc(e.f, p)`, the consumption tries to find a chunk $t_{\mathsf{e}}.\mathtt{f} \mapsto \_ \# t_{\mathsf{q}}$ on the heap and subtracts $t_{\mathsf{p}}$ from $t_{\mathsf{q}}$. If the heap chunk is then empty, i.e. $\sigma \vdash_{\mathsf{z3}} t_{\mathsf{q}} - t_{\mathsf{p}} = 0$, it is removed from the heap. In the case that a heap chunk is modified, the snapshot term (read here: the value) of the modified chunk is returned. Therefore, consuming an assertion can affect the current heap and the path conditions, but the other state components will remain unaltered.

Assertions are consumed by the function

- **consume** : $\Sigma \times T_\alpha \times \Phi \times (\Sigma \times T \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$, where the second argument is a factor used to scale the access permissions that are to be consumed[9]. **consume** is defined with the helper function
- **consume'** : $\Sigma \times H \times T_\alpha \times \Phi \times (H \times \Pi \times T \rightarrow \mathit{Bool}) \rightarrow \mathit{Bool}$, which passes the remaining heap, newly gained path conditions and the consumed snapshot to the continuation. The additional heap argument (second argument) represents the remaining

[6] The third parameter in [Sch11], the currently evaluated functions, has since been removed.
[7] Snapshots are used to frame predicate and function applications. They summarise the field values at a given state. They are orthogonal to this project.
[8] This is due to the fact that predicates are also heap objects, and is not relevant for this project.
[9] Again, this is due to the fact that predicates are also heap objects, and is not relevant for this project.

heap which can be modified, and which is eventually passed to the continuation. The state (first argument) is used to evaluate expressions.

The three functions introduced so far – **eval**, **produce**, and **consume** – all handle assertions or expressions, but do not cover statements are thus not suited to actually execute programs.

Statements are symbolically executed by

**exec** : $\Sigma \times \mathcal{S} \times (\Sigma \to \mathit{Bool}) \to \mathit{Bool}$ , which executes a single statement. It is implemented with **eval**, **produce**, and **consume**, and affects all components of the state.

Based on these functions, the validity of program functions, methods, and consequently, a program, is defined. A slightly outdated version of these rules is presented in [Sch11]. In particular, Silicon by now supports the evaluation of quantifiers with pure bodies. So, we can focus on quantifiers with access predicates in the body.

### 2.2.3 Sets and sequences

All set and sequence operations in SIL have a direct corresponding term: e.g., getting a sequence with the first `n` items of sequence `S` dropped, `S[n..]`, is $t_{\mathtt{S}}[t_{\mathtt{n}}..]$ as a term. As sets and sequences are not natively supported by our SMT solver, assertions about set and sequence operations would not verify without additional knowledge about the semantics of the operations. Hence, to prove assertions about sets and sequences, Silicon comes with an axiomatisation of these operations.

# 3 Permissions under quantifiers

## 3.1 Semantics

In general, a universal quantification has the form

```
forall x1:T1, ..., xn:Tn :: Q
```

with quantified variables `x1`, ..., `xn` of types `T1`, ..., `Tn` and expression `Q`. Such a quantification can be expressed equivalently as a (possibly infinite) conjunction of its instantiations. We define this conjunction to be the separating conjunction. For example,

```
forall i:Int :: i in [0..2] ==> acc(Seq(x,x)[i].f, 1/2)
```

is semantically equivalent to `acc(x.f, 1)`.

The fragment of IDF supported by Silicon forbids access permissions on the left side of implications. This allows us to deduce that quantifiers that mix access assertions and pure expressions are actually syntactic sugar, and can be separated into pure quantifiers and permissions under quantifiers with a special form.

### 3.1.1 Canonical quantifers

We separate pure and access assertions.

**Definition 11** (Canonical quantifier)**.** A *canonical quantifier* has the form

```
forall x1:T1, ..., xn:Tn :: Q
```

where `Q` is either a pure boolean expression or has the form

```
P ==> acc(e.f, p)
```

where `P` is a pure boolean expression. Here, `e` evaluates to the receiver, `f` $\in \mathcal{F}$ is a field, and `p` is an expression that evaluates to a fractional permission. ⌟

**Lemma 1.** *Each quantifier* `forall x1:T1, ..., xn:Tn :: Q` *is equivalent to a conjunction of canonical quantifiers.*

*Proof.* Our fragment of IDF is defined as follows:

```
expr :  pure | acc(pure.f, pure) | expr && expr | pure ==> expr
```

where `pure` is a pure expression, i.e. it does not contain access predicates, and `f` is a field. We assume that every expression is typed correctly. No access predicate may appear on the left side of an implication. We do not consider nested quantifiers. Without loss of generality, we only use one quantified variable.

We proceed by giving a tuple $(n, m)$ to each quantifier `forall x:T :: Q`, where $n$ is the number of implications (`==>`) in `Q`, and $m$ is the number of conjunctions (`&&`) in `Q`. We only count implications and conjunctions when at least one side (for implications, it is always the right side) is not pure.

Our proof is by lexicographic induction on the tuple $(n, m)$. We have two induction bases:

1. $(0, 0)$: We case-split on the structure of `Q`.
   a) `Q = P`, where `P` is pure. `forall x:T :: P` is a canonical quantifier (and therefore equivalent to a conjunction of canonical quantifiers).
   b) `Q = acc(P1.f, P2)`, where `P1` and `P2` are pure. An implication `true ==> Q` is equivalent to `Q`. Therefore, `forall x:T :: true ==> acc(P1.f, P2)` is the equivalent canonical quantifier.

   There are no other quantifiers with no implication and no conjunction.
2. $(1, 0)$: We case-split on the structure of `Q`.
   a) `Q = P1 ==> P2`, where `P1` and `P2` are pure. `forall x:T :: P1 ==> P2` is a canonical quantifier.
   b) `Q = P1 ==> acc(P2.f, P3)`, where `P1`, `P2`, and `P3` are pure. `forall x:T :: P1 ==> acc(P2.f, P3)` is a canonical quantifier.

   There are no other quantifiers with one implication and no conjunction.

For the induction step, we assume now that `Q` contains $n$ implications and $m$ conjunctions. To apply the induction hypothesis, we need to reduce the quantifier to a conjunction of quantifiers for which each contains either fewer conjunctions and the same number of implications, or fewer implications. We case-split on the structure of `Q`.

1. `Q = E1 && E2`, where `E1` and `E2` are expressions. We rewrite to a equivalent conjunction of two quantifiers, `forall x:T :: E1 && forall x:T :: E2`. Each of these conjuncts has $n$ implications, and $m - 1$ conjunctions, and is thus by the induction hypothesis equivalent to a conjunction of canonical quantifiers.
2. `Q = P1 ==> E1 && E2`, where `P1` is pure, and `E1` and `E2` are expressions. We rewrite to `forall x:T :: P1 ==> E1 && forall x:T :: P1 ==> E2`. Each of these conjuncts has $n$ implications, and $m - 1$ conjunctions. We apply the induction hypothesis for both conjuncts.

3. `Q = P1 ==> P2 ==> E1`, where `P1` and `P2` are pure, and `E1` is an expression. We rewrite to `forall x:T :: (P1 && P2) ==> E1`, which has $n-1$ implications, and $m$ conjunctions (both sides of `P1 && P2` are pure). We apply the induction hypothesis.

□

Hence, it is enough to consider canonical quantifiers, as all other quantifiers can syntactically be rewritten to conjunctions of canonical quantifiers. The proof constitutes an algorithm for this rewriting. As we have shown that pure quantifiers can be separated from permissions under quantifiers, and Silicon supports pure quantifiers already, we can now focus our efforts on access permissions under quantifiers.

## 3.2 Introductory examples

Permissions under quantifiers can be used to create quite complex heap structures. They introduce logic to the heap structure. As Silicon stores the heap structure separate from the prover, also this logic is separated from the prover. The following examples introduce some challenges that we will have to deal with.

### 3.2.1 Sets

Listing 3.1 shows how we can represent access to a set of objects with permissions under quantifiers. The precondition of method `sets` requires write access to all elements in `S`. Because it is known that the element `a` is in `S`, it should be possible to write to the location `a.f`. For another element `b`, the value of `b.f` should not change.

A more intricate example is method `sets_intersect`. If we have half permission to elements in `S` and half permission to elements in `T`, we should have full access to all elements in `S intersection T`.

Listing 3.1: The access to field locations where the receiver is in a set can be represented with permissions under quantifiers.

```
1  var f:Int
2
3  method sets(S:Set[Ref], a:Ref, b:Ref)
4  requires forall s:Ref :: s in S ==> acc(s.f, write)
5  requires a in S && b in S && b != a
6  ensures forall s:Ref :: s in S ==> acc(s.f, write)
7  ensures b.f == old(b.f)
8  {
9          a.f := 5
10 }
11
12 method sets_intersect(S:Set[Ref], T:Set[Ref], a:Ref)
13 requires forall s:Ref :: s in S ==> acc(s.f, write)
14 requires forall t:Ref :: t in T ==> acc(t.f, write)
15 requires a in S && a in T
```

```
16 ensures forall x:Ref :: x in (S intersection T) ==> acc(x.f, write)
17 {
18         a.f := 5
19 }
```

### 3.2.2 Sequences

Listing 3.2 shows how we can represent access to a sequence of objects. Permissions are given for each index. Note that the objects at each index are not necessarily distinct. That is why, in the example, if two indices alias, write permission is held. So, an assertion for sequences can inhale permission to a specific location more than once if objects in the sequence alias. Additionally, if in the example `S[0].f` would be written, `S[1].f` would also change.

Listing 3.2: Permissions are inhaled for each index of a sequence.

```
1 var f: Int
2
3 method sequences(S:Seq[Ref])
4 requires forall i:Int :: i in [0..|S|] ==> acc(S[i].f, 1/2)
5 requires |S| > 2 && S[0]==S[1]
6 {
7         // should work because half permission has been inhaled for indices
              0 and 1
8         exhale acc(S[0].f, write)
9 }
```

### 3.2.3 Modelling arrays with sequences

Quantified permissions allow one to model the behavior of arrays with sequences. The idea is to keep a sequence of *location* objects: each of these location objects models one cell of the array. The object at each index holds the value in a designated field. Then, we can distribute permission to access these field locations. It is important that these location objects are mutually distinct: if one index is written, the values at other indices should not be changed. We gain this distinctness of location objects implicitly if write permission for each index is held.

Listing 3.3 illustrates this idea. From the distinctness of the cells the assertion `S[b].f == old(S[b].f)` can be established.

Listing 3.3: The behavior of arrays can be modelled with sequences and quantified permissions.

```
1 var f:Int
2
3 method array(S:Seq[Ref],a:Int, b:Int)
4 requires forall i:Int :: i in [0..|S|) ==> acc(S[i].f, write)
```

```
5  requires a >= 0 && b > a && b < |S|
6  {
7          S[a].f := 5
8          // implicitly, we know that S[a] != S[b]
9          assert S[b].f == old(S[b].f)
10 }
```

# 4 Quantified Chunks

We have seen that Silicon uses field chunks of the form $t_r.\mathtt{f} \mapsto t_v \# t_\alpha$ to represent the current state of a field location. If such a chunk exists in the heap at some program point, it is known that permission $t_\alpha$ is held for location $t_x.\mathtt{f}$, and its value is $t_v$ at that point. For two field locations, two heap chunks can be put on the heap, three for three fields locations, and so on. However, in the case of permissions under quantifiers, we need to represent the access to an arbitrary number of field locations, for example for all elements of a set. Therefore, we introduce *quantified chunks*, which generalise the idea of fields chunks to an arbitrary number of field locations.

## 4.1 Form and semantics

**Definition 12.** A **quantified field chunk** has the form $\forall x \bullet x.\mathtt{f} \mapsto t_v \# t_\alpha$ where

- $x \in \mathit{Ref}$ is the receiver object that is being quantified over
- $\mathtt{f} \in \mathcal{F}$ is a field name
- $t_v \in T$ is a term that represents the field value
- $t_\alpha \in T_\alpha$ is a fractional permission term representing how much permission to access the location is given

We define $QCH$ to be the set of quantified field chunks, a subset of $CH$. ⌟

If such a quantified field chunk exists on the heap, permission $t_\alpha$ is held for the field $\mathtt{f}$ of any non-null object $x$. If then additionally $t_\alpha > 0$, the value of $x.\mathtt{f}$ is currently $t_v$. Else, $t_\alpha = 0$, and the value of $x.\mathtt{f}$ may not be read. Note that both $t_\alpha$ and $t_v$ are terms and usually depend on $x$.

**Example 1.** A chunk $\forall x \bullet x.\mathtt{f} \mapsto t(x) \# x \in S \; ? \; 1 \; : \; 0$ on the heap entails that at this program point, full permission is held for the field $\mathtt{f}$ of all objects in set $S$. Let $y$ be an object with $y \in S$. As $(y \in S \; ? \; 1 \; : \; 0)$ is then strictly positive, the value of $y.\mathtt{f}$ is $t(y)$, where $t$ is a function from objects to $\mathit{sort}(\mathtt{f})$. ⌟

As this example shows, quantified field chunks can be used to represent the permission to a set of fields.

Quantified field chunks subsume "original" field chunks. A field chunk $t_x.f \mapsto t_v \# t_\alpha$ can be written equivalently as quantified field chunk

$$\forall x \bullet x.\mathtt{f} \mapsto t_v \# (x = t_x \; ? \; 1 \; : \; 0) * t_\alpha.$$

$$\textbf{quantifyChunksForField}(h, \mathtt{f}) = \textbf{map}(h, (\lambda\, ch \bullet$$
$$\quad \textbf{if}\ ch = t_x.\mathtt{f} \mapsto t_v \,\#\, t_\alpha$$
$$\quad\quad \forall x \bullet x.\mathtt{f} \mapsto t_v \,\#\, x = t_x \;?\; t_\alpha \;:\; 0$$
$$\quad \textbf{else}\ ch))$$

$$\textbf{isQuantifiedForField}(h, \mathtt{f}) = \exists\, ch \in h \bullet ch = \forall x \bullet x.\mathtt{f} \mapsto \_ \,\#\, \_$$

where **map** is the higher-order mapping function as e.g. available in Haskell with the first argument being the data structure to iterate over, and the second argument being the mapping function.

Figure 4.1: Non-quantified field chunks can be converted to quantified chunks.

Note that the value $t_v$ does not have to depend on $x$. Especially when converting non-quantified chunks to quantified ones, $t_\alpha > 0$ for exactly one receiver, and so $t_v$ can as well be a literal of $sort(\mathtt{f})$.

With this conversion, we do not have to worry about different types of chunks on the heap for a field: in particular, it allows us to stick to the simpler operations on non-quantified chunks as long as there are no quantified chunks on the heap. As soon as a quantified chunk is placed on the heap, all non-quantified chunks are converted using the transformation above. From then on, only the operations on quantified chunks are used.

Moreover, observe that each field can be considered separately, as all operations on chunks only concern a single field and fields are never mixed. Therefore, while some fields operate with quantified chunks (we call them *quantified fields*), for the other fields the simpler operations for non-quantified chunks can be used.

Based on this conversion, two functions are defined in Figure 4.1:

- **quantifyChunksForField** : $H \times \mathcal{F} \to H$

  takes a heap and a field name and applies the conversion to all non-quantified heap chunks for the given field name.

- **isQuantifiedForField** : $H \times \mathcal{F} \to \mathit{Bool}$

  takes a heap and a field name and checks if the heap operates with quantified chunks for the given field name. This function helps to decide if operations on quantified chunks should be applied or the operations on non-quantified chunks suffice.

We will discover the subtleties of the semantics of quantified chunks while defining the operations on them.

$$\begin{aligned}&\textbf{permission}(h, t_x, \mathtt{f}) = \textbf{fold}(h, 0, (\lambda\ sum, ch \bullet\\ &\quad \textbf{if}\ ch = (\forall x \bullet x.\mathtt{f} \mapsto \_ \ \#\ t_\alpha)\\ &\qquad sum + t_\alpha[x \to t_x]\\ &\quad \textbf{else}\ sum))\end{aligned}$$

where **fold** is the folding higher-order function as e.g. available in Haskell with the first argument being the data structure to iterate over, the second argument being the initial accumulator, and the third argument being the combining function.

Figure 4.2: The permission held for a specific location is the symbolic sum over all permissions of chunks for that field name.

## 4.2 Operations on quantified chunks

### 4.2.1 Calculating the permissions held for a field location

When accessing a field location, we need to prove that we have positive permission to that location. When writing a field location, we need to prove that we have write permission to that location. Thus, we need an operation to get the permission for a location.

The permission for a field may be distributed among multiple chunks for the same field name:

**Example 2.** (Recall Listing 3.1) If chunks

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \ \#\ x \in T\ ?\ 0.5\ :\ 0 \qquad\qquad \forall x \bullet x.\mathtt{f} \mapsto \_ \ \#\ x \in S\ ?\ 0.5\ :\ 0$$

exist on the heap, full permission is held for any location $t_x.\mathtt{f}$ where $t_x \in T \cap S$. ⌟

In general, the permission held for a location is the symbolic sum over all permissions with the field name, instantiated with the receiver. In the previous example, this is

$$(t_x \in T\ ?\ 0.5\ :\ 0) + (t_x \in S\ ?\ 0.5\ :\ 0)$$

which, as we know that $t_x \in T \cap S$, can be proved to be equivalent to $1$. This insight is captured in the function

$$\textbf{permission} : H \times T_R \times \mathcal{F} \to T_\alpha$$

which is implemented in Figure 4.2. It takes a heap, an object term, and a field name, and returns a term that represents the permissions for the field location.

### 4.2.2 Exhaling permissions

The **permission** function defined above already allows us to decide if we have enough permission to exhale. For example, if we want to exhale full permission to $x.\mathtt{f}$ for all $x \in S$,

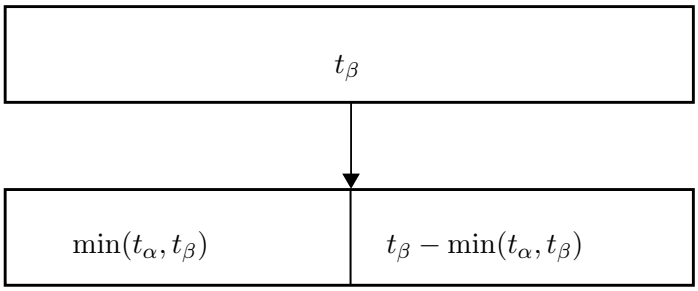



Figure 4.3: In one step of **exhale**, the permissions to be exhaled are split into two parts: the permissions gained by the current chunk, and the permissions that remain to be exhaled.

we take a fresh object variable $t_x$, assume $t_x \in S$, and try to prove **permission**$(h, t_x, \mathtt{f}) = 1$. Still, what needs to be done is to calculate the modified heap after the exhale.

Observe that the required permissions might not reside in a single chunk: we might have to exhale permissions for all $x \in T \cup S$ although we have separate chunks for $x \in S$ and $x \in T$. Moreover, some elements might be in $x \in S \cap T$, while others are not.

Consider the situation where we have only one chunk

$$ch_\alpha = \forall x \bullet x.\mathtt{f} \mapsto \_ \# t_\alpha$$

on the heap. Let

$$ch_\beta = \forall x \bullet x.\mathtt{f} \mapsto \_ \# t_\beta$$

be a chunk representation of what we want to exhale. Let $\min(t_\alpha, t_\beta)$ be the minimum of the permission terms. Formally, $\min(t_\alpha, t_\beta)$ is an abbreviation for the term $t_\alpha < t_\beta \,?\, t_\alpha \,:\, t_\beta$. Intuitively, $\min(t_\alpha, t_\beta)$ represents "what we can get" from $ch_\alpha$ when trying to exhale $ch_\beta$ So, if we now analyse $t_\beta - \min(t_\alpha, t_\beta)$, there are two cases: (1) $\forall\, x \bullet t_\beta - \min(t_\alpha, t_\beta) = 0$. This implies that $ch_\alpha$ contains all the permissions in $ch_\beta$. (2) else, $\exists\, x \bullet t_\beta - \min(t_\alpha, t_\beta) \neq 0$. This implies that there are still permissions left to exhale.

Figure 4.3 illustrates this process: Intuitively, we split chunk $ch_\beta$ in two parts: those permissions that can be gained from $ch_\alpha$, which is $\min(t_\alpha, t_\beta)$, and those that remain, $t_\beta - \min(t_\alpha, t_\beta)$. The case distinction above decides whether the exhale for this one-chunk-heap succeeded or failed. In the case the exhale succeeded, we have to subtract from $ch_\alpha$ the permissions we took from it. Hence, the resulting heap would be

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \# t_\alpha - \min(t_\alpha, t_\beta)$$

Note that we can check whether or not $t_\alpha - \min(t_\alpha, t_\beta) = 0$ and remove the chunk from the heap if it carries no more permission.

The function

**exhale** : $\Sigma \times H \times QCH \times (H \to Bool) \to Bool$

generalises the discussed process for multiple chunks. Its first argument is the current state, its second the initial heap, its third is a chunk representation of the permissions to exhale, and the fourth argument is a continuation to which the resulting heap is passed if the exhale succeeds. It is implemented in Figure 4.4.

The implementation folds over the chunks in the original heap. The accumulator holds three components: the chunks representing the permission that is left to exhale, the modified heap, and a flag that indicates whether or not the exhale is already done. One call to the combining function receives the current accumulator and a heap chunk, and proceeds as follows:

1. If the exhale is already done or the heap chunk is for a different field, it continues with the accumulator with the heap chunk added. This is to make sure that all unmodified chunks are kept.
2. Else, the step is similar to the one we described above: the minimum of the permissions is subtracted from both the chunk of the heap and the chunk that contains what we want to exhale. The function then checks whether enough permissions have already been exhaled: if this is the case, the flag `done` is set and the rest of the chunks are passed through unmodified. Otherwise, the function continues exhaling with the subtracted permission. The heap chunk that we took permissions from is added if it still carries permissions.

The exhale succeeds when at some point no permissions remain to exhale.

**Example 3.** We have the two chunks

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \# x \in t_S \setminus \{t_x\} \ ?\ 1\ :\ 0 \qquad\qquad \forall x \bullet x.\mathtt{f} \mapsto \_ \# x = t_x \ ?\ 0.5\ :\ 0$$

on the heap. We want to exhale half permission to elements in $t_S$, i.e.

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \# x \in t_S \ ?\ 0.5\ :\ 0$$

We gain $\alpha := \min\left((x \in t_S)\ ?\ 0.5\ :\ 0, (x \in t_S \setminus \{t_x\})\ ?\ 1\ :\ 0\right)$ from the first chunk. What remains in the chunk is $(x \in t_S \setminus \{t_x\}\ ?\ 1\ :\ 0) - \alpha$, which can be simplified to $x \in t_S \setminus \{t_x\}\ ?\ 0.5\ :\ 0$. What is still left to exhale is $(x \in t_S\ ?\ 0.5\ :\ 0) - \alpha$, which can be simplified to $(x = t_x\ ?\ 0.5\ :\ 0)$.

From the second chunk, we thus gain $\beta := \min(x = t_x\ ?\ 0.5\ :\ 0, x = t_x\ ?\ 0.5\ :\ 0)$. The second chunk is then empty, because $(x = t_x\ ?\ 0.5\ :\ 0) - \beta$ can be simplified to $0$, and is thus removed. For the same reason, but in a different prover call, the remaining permission $(x = t_x\ ?\ 0.5\ :\ 0) - \beta$ is shown to be $0$, and we are done and the exhale succeeded. A single chunk

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \# x \in t_S \setminus \{t_x\} \ ?\ 0.5\ :\ 0$$

remains on the heap. ⌟

Note that when exhaling, the order of the chunks has an impact on the performance: If the chunks that actually contain the permissions come first, prover calls can be avoided and

$$
\begin{array}{l}
\textbf{exhale}(\sigma, h, ch, Q) = \\
\quad \textbf{let} \\
\quad\quad (\_, h', success) = \textbf{foldl}(h, (ch, \emptyset, false), (\lambda(ch_\beta, h_1, done), ch_\alpha \bullet \\
\quad\quad\quad \textbf{if } done \\
\quad\quad\quad\quad (ch_\beta, h_1 + ch_\alpha, done) \\
\quad\quad\quad \textbf{else if } ch_\alpha = \forall x \bullet x.\mathtt{f} \mapsto \_ \,\#\, t_\alpha \ \wedge ch_\beta = \forall x \bullet x.\mathtt{f} \mapsto \_ \,\#\, t_\beta \\
\quad\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad\quad r = min(t_\alpha, t_\beta),\ d = \sigma \vdash_{\mathsf{z3}} (\forall\, x \bullet t_\beta - r = 0), \\
\quad\quad\quad\quad\quad ch'_\alpha = \forall x \bullet x.\mathtt{f} \mapsto \_ \,\#\, t_\alpha - r, \\
\quad\quad\quad\quad\quad ch'_\beta = \forall x \bullet x.\mathtt{f} \mapsto \_ \,\#\, t_\beta - r \\
\quad\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad\quad \textbf{if } \sigma \vdash_{\mathsf{z3}} (\forall\, x \bullet t_\alpha - r = 0) \\
\quad\quad\quad\quad\quad\quad (ch'_\beta, h_1, d) \\
\quad\quad\quad\quad\quad \textbf{else } (ch'_\beta, h_1 + ch'_\alpha, d) \\
\quad\quad\quad \textbf{else } (ch_\beta, h_1 + ch_\alpha, false))) \\
\quad \textbf{in } success \wedge Q(h')
\end{array}
$$

Figure 4.4: **exhale** implements exhaling quantified chunks.

expressions stay simpler. The order of the chunks could be controlled by heuristics, e.g. when syntactically identical variables are mentioned (modulo known equalities). We have not implemented such an optimisation.

### 4.2.3 Getting values

Until now, we ignored the values of these chunks. As the permission to access a field of a receiver can be spread over multiple chunks, so can the information about the values:

- It might occur that the permission is greater than zero for the same receiver for more than one chunk for the same field name. As a location can only have one symbolic value at a program point, all these values need to be equal:

  **Example 4.** We have the two chunks

  $$\forall x \bullet x.\mathtt{f} \mapsto t_v(x) \# x = t_x \ ?\ 0.5 \ :\ 0 \qquad \forall x \bullet x.\mathtt{f} \mapsto 5 \# x = t_x \ ?\ 0.5 \ :\ 0$$

  on the heap. The current value of $t_x.\mathtt{f}$ is $t_v(t_x) = 5$. ⌟

- The values for different receivers in the same set might reside in two or more chunks:

  **Example 5.** We have the two chunks

  $$\forall x \bullet x.\mathtt{f} \mapsto t_v(x) \# x \in S \setminus \{t_x\} \ ?\ 0.5 \ :\ 0 \qquad \forall x \bullet x.\mathtt{f} \mapsto 5 \# x = t_x \ ?\ 0.5 \ :\ 0$$

  on the heap. The value of $t_y.\mathtt{f}$ with $t_y \in S$ is $t_y.\mathtt{f} = \begin{cases} 5 & \text{if } t_y = t_x \\ t_v(t_y) & \text{if } t_y \in S \setminus t_x \end{cases}$. ⌟

Both issues can be solved by using *field value summaries*. The idea is to create a fresh function that represents all the knowledge contained in the heap about the values for a certain field at a program point.

**Definition 13.** A *field value summary* for field name $\mathtt{f}$ and heap $h$ is a fresh function $t$ from objects to $sort(\mathtt{f})$. It is axiomatised by the contents of the heap for field $\mathtt{f}$: for each chunk $\forall x \bullet x.\mathtt{f} \mapsto t_v \# t_\alpha$ in the heap for field $\mathtt{f}$, an axiom partially specifying $t$ is added:

$$\forall\, x \bullet t_\alpha > 0 \implies t(x) = t_v$$

⌟

The field value is accordingly represented by the field value summary applied at the receiver.

**Example 6.** (continuing Example 4) The field value summary $t$ for field $\mathtt{f}$ has the properties

$$\forall\, x \bullet (x = t_x \ ?\ 0.5 \ :\ 0) > 0 \implies t(x) = t_v(x)$$
$$\forall\, x \bullet (x = t_x \ ?\ 0.5 \ :\ 0) > 0 \implies t(x) = 5$$

Thus, $t(t_x) = t_v(t_x) = 5$. ⌟

$$
\begin{aligned}
&\textbf{value}(\sigma, t_x, \mathtt{f}, Q) = \\
&\quad \textbf{let} \\
&\qquad t = \mathit{fresh}_{\mathit{Ref}\to \mathit{sort}(\mathtt{f})}, \\
&\qquad \pi' = \textbf{fold}(\sigma.h, \emptyset, (\lambda\, \pi_1, ch \bullet \\
&\qquad\quad \textbf{if } ch = (\forall x \bullet x.\mathtt{f} \mapsto t_v \,\#\, t_\alpha) \\
&\qquad\qquad \pi_1 + (\forall\, x \bullet t_\alpha > 0 \implies t(x) = t_v) \\
&\qquad\quad \textbf{else } \pi_1)) \\
&\quad \textbf{in} \\
&\qquad \sigma \vdash_{\mathsf{z3}} t_x \neq \mathrm{null} \ \wedge \sigma \vdash_{\mathsf{z3}} \textbf{permission}(h, t_x, \mathtt{f}) > 0 \ \wedge Q(\sigma.\pi + \pi', t(t_x))
\end{aligned}
$$

where $t(x)$ denotes a symbolic function application.

Figure 4.5: **value** creates a field value summary for the given field.

**Example 7.** (continuing Example 5) The field value summary of $t$ for field $\mathtt{f}$ has the properties

$$\forall\, x \bullet (x \in S \setminus \{t_x\}\ ?\ 0.5\ :\ 0) > 0 \implies t(x) = t_v(x)$$
$$\forall\, x \bullet (x = t_x\ ?\ 0.5\ :\ 0) > 0 \implies t(x) = 5$$

Thus, $t(t_y) = \begin{cases} 5 & \text{if } t_y = t_x \\ t_v(t_y) & \text{if } t_y \in S \setminus \{t_x\} \end{cases}$ ⌟

Function

$$\textbf{value} : \Sigma \times T_R \times \mathcal{F} \times (\Pi \times T \to \mathit{Bool}) \to \mathit{Bool}$$

implements field value summaries in Figure 4.5. Its first argument is the current state, the second a receiver, third a field name, fourth the continuation to which the updated path conditions and the new heap value summary at the receiver are passed. To ensure consistency, it additionally succeeds only if the receiver is non-null and there is permission to read the field location at all.

Some ideas could be experimented with to optimise how values are treated:

- If the permission to access a location resides only in one chunk, the value of this chunk can be directly returned. (watch out for situations such as Example 4 though).
- **value** creates a fresh function each time it is called, and thus in particular for each field access. It might be possible to cache these functions if the relevant heap chunks have not changed. This could reduce the number of introduced functions and improve the performance.
- Most often, the permission for a receiver resides in a subset of the chunks for a specific field. It might be enough to axiomatise the summary only with the values of this

subset, hence creating a *partial field summary*. A subset (though not necessarily the minimal one) can be found with the exhale algorithm.

## 4.3 From quantified assertions to quantified chunks

We have already seen some quantified chunks. However, it is still unclear how a quantified assertion

```
forall y:T :: P ==> acc(e.f, p)
```

can be represented as a quantified chunk

$$\forall x \bullet x.\mathtt{f} \mapsto t_v \# t_\alpha$$

on the heap.

Our idea is to give the chunk's permission the form $t_\alpha = count * t_p$, where $count$ is a term that depends on $x$, and $t_p$ is the evaluated permission. $count$ is a term that yields the number of instantiations of y such that e evaluates to $x$. Each such instantiation inhales $t_p$ permission to $x.\mathtt{f}$, and thus $count$ needs to be multiplied with $t_p$. This is why we only need a single quantified variable in the chunk and it is always of sort *Ref*: The idea of $count$ is that it condenses the instantiations of y into a term that yields the number of evaluations of expression e for which $x$ is "hit" by the quantifier.

**Example 8.** When a simple chunk $t_x.\mathtt{f} \mapsto \_ \# t_p$ is converted, the converted chunk has the form $\forall x \bullet x.\mathtt{f} \mapsto \_ \# (x = t_x \ ? \ 1 \ : \ 0) * t_p$. The permission $x = t_x \ ? \ 1 \ : \ 0$ is the term that counts how often permission $t_p$ is inhaled: exactly once, when $x = t_x$. This is a special case of the set idiom introduced in subsection 4.3.1. ⌟

As a general encoding of $count$ seems challenging[1], the idea is to partition the assertions into subsets, called *idioms*, and handle them separately. For each of these idioms, rules for two transformation functions are given.

- **transform** : $T_R \times \mathcal{F} \times T_{Bool} \times T \times T_\alpha \to QCH$
  which takes a receiver, a field name, a guard, a value, and a permission, and returns a quantified chunk.
- **transformElement** : $T_R \times \mathcal{F} \times T \times T_\alpha \to QCH$
  takes the same arguments, but without the guard. It is used to rewrite non-quantified assertions.

The implementation of these functions for the two supported idioms, sets and splitting sequences, is given in Figure 4.6 and Figure 4.7; the rationale of the rules is explained in

[1] In general, one would have to axiomatise a term $count$ that depends on the receiver $x$ and yields how often e evaluates to $x$ with the guard being true. Formally, $count(x) = |\{(\mathsf{y}|e_\mathsf{y} = x \wedge t_{guard}\}|$. We suspect that there is no simple, general first-order logic definition or axiomatisation of such a term, but made no further effort to support this claim.

**transform**$(t, \mathtt{f}, guard, t_v, t_\alpha) =$
$\forall x \bullet x.f \mapsto t_v[t \to x] \# (guard[t \to x] \; ? \; 1 \; : \; 0) * t_\alpha[t \to x]$
where $t$ is a symbolic object literal of the variable the assertion quantifies over.

**transform**$(S[i], \mathtt{f}, i \in [start..end], t_v, t_\alpha) =$
$\forall x \bullet x.f \mapsto t_v[S[i] \to x] \# count(S[start..end], x) * t_\alpha[S[i] \to x]$
where $i$ is a symbolic integer literal of the variable the assertion quantifies over,
and $S[i]$ is a sequence access.

Figure 4.6: **transform** takes a receiver, a field name, the guard, a value, and a permission, and returns the chunk transformed by the corresponding rule.

**transformElement**$(t, \mathtt{f}, t_v, t_\alpha) =$
**let** $s = fresh_{Ref}$ **in transform**$(s, \mathtt{f}, s = t, t_v, t_\alpha)$

**transformElement**$(S[i], \mathtt{f}, t_v, t_\alpha) =$
**let** $j = fresh_{Int}$ **in transform**$(S[j], \mathtt{f}, j \in [i..i+1], t_v, t_\alpha)$

Figure 4.7: **transformElement** takes a receiver, a field name, a value, and a permission, and returns the chunk transformed by the corresponding rule.

the following sections. In addition, the required axiomatisation of functions that appear in the transformation is given.

Note that we chose to rewrite on terms instead of expressions. There are two other options to be considered: (1) One could take an expression, evaluate it to a term, and return a chunk. This would have required a call to **eval** in the implementation of **transform**, which we chose to leave to the symbolic execution rules. (2) One could directly rewrite the AST. However, the transformation would include expressions that are technical and not meant to be exposed as syntax.

A disadvantage of the separation into idioms is that these idioms cannot be fully mixed. This incompleteness is described in subsection 5.3.2.

### 4.3.1 Sets

This idiom handles all assertions which have the quantified variable as receiver. This ensures that permission is inhaled either once per receiver, or not at all.

#### Supported assertions

We support all assertions of the form

```
forall x:Ref :: guard ==> acc(x.f, p)
```

where guard is pure. The assertion has to quantify over one variable of reference type, which is also the receiver.

#### Transformation

Let $t_{guard}$, $t_x$, and $t_p$ be the evaluated expressions in the current state. Permission $t_p$ is inhaled to $t_x.\mathtt{f}$ iff the guard is true. Hence, the permission of the chunk is $(t_{guard}\ ?\ 1\ :\ 0)*t_\alpha$.

For single elements $t_y$, we let the guard be $x = t_y$, and transform with this new guard.

**Example 9.** The assertion

```
forall x:Ref :: x in S ==> acc(x.f, 1/2)
```

is represented as chunk $\forall x \bullet x.\mathtt{f} \mapsto t_v \ \#\ (x \in t_\mathsf{S}\ ?\ 1\ :\ 0) * \frac{1}{2}$, where $t_\mathsf{S}$ is the symbolic value of S in the current state, and $t_v$ is the value given as the fourth parameter to **transform**. The value depends on the concrete use case of the transformation, e.g. inhaling or exhaling.

The assertion

```
acc(y.f, write)
```

is represented as chunk $\forall x \bullet x.\mathtt{f} \mapsto t_v \ \#\ (x = t_\mathsf{y}\ ?\ 1\ :\ 0)$, where $t_\mathsf{y}$ is the symbolic value of y in the current state, and $t_v$ is the third parameter given to **transformElement**. Again, the value depends on the concrete use case of the transformation. ⌟

#### Axiomatisation

As this transformation introduces no additional functions, no axioms are needed.

### 4.3.2 Splitting sequences

Sets have the inherent property that their contained objects are distinct. This no longer holds true for sequences: in general, the same object might be stored at different indices of a sequence. An assertion can thus inhale permission to the same location more than once.

To model this, we introduce a counting function and axiomatise it accordingly. We focus on typical array usage scenarios: writing values, reading values, and splitting a sequence into a number of parts, e.g. to treat each part in a separate thread. Our axiomatisation is designed to split the sequence into parts where each part is expressed explicitly as a subrange, i.e. from indices $start$ to $end$.

Our axiomatisation is not complete for algorithms that need to split the permissions in a more general way: For example, there might be algorithms that need to split permissions to a sequence into even and odd indices. Our axiomatisation does not support this yet. However, it works well for typical array usage scenarios, e.g. mergesort.

#### Supported assertions

Assertions of the form

```
forall i:Int :: i in [a..b] ==> acc(S[i].f, p)
```

are supported, i.e. the assertion quantifies over one variable of integer type, the guard constrains the quantified variable to be in a range, and the receiver is a sequence that is accessed at the quantified variable, with `f` being a field name. The integer expressions `a` and `b` can be arbitrary but must not depend on `i`.

#### Transformation

Let $t_\mathsf{a}$, $t_\mathsf{b}$, $t_\mathsf{p}$, $t_\mathsf{S}$ be the terms resulting from the evaluation of `a`, `b`, `p`, `S`, respectively, in the current state. Permission $t_\mathsf{p}$ is inhaled to the object at each index. We introduce the (uninterpreted) function

$$count : Seq \times Ref \to Int$$

which takes a sequence as well as an object and represents the count of how often this object is contained in the sequence. Intuitively, if $count(t_S, x) = n$, then there are $n$ mutually distinct indices $0 \leq i_1, \ldots, i_n < |t_\mathsf{S}|$ for which $t_\mathsf{S}[i_1] = \cdots = t_\mathsf{S}[i_n] = x$. However, precisely formulating this function in a SMT solver is challenging[2]. Therefore, we leave this function uninterpreted and only give the required axioms to fulfil common proof obligations. The

[2] We could not come up with a non-recursive first-order logic formulation. A recursive formulation is possible ($count(S[a..b], x) = (S[a] = x \ ? \ 1 \ : \ 0) + count(S[a+1..b], x)$), but for an arbitrary set, it is unclear when to terminate the recursion. Inductive proofs are not supported by Z3.

transformation produces a permission term

$$count(t_{\mathsf{s}}[t_{\mathsf{a}}..t_{\mathsf{b}}], x) * t_{\mathsf{p}}.$$

For single elements $t_{\mathsf{s}}[i]$, we let the guard be $j \in [i..i+1]$, and transform with this new guard.

**Example 10.** The assertion

```
forall i:Int :: i in [0..|S|] ==> acc(S[i].f, write)
```

is represented as chunk $\forall x \bullet x.\mathsf{f} \mapsto t_v \# count(t_{\mathsf{s}}[0..|t_{\mathsf{s}}|], x)$. The value $t_v$ depends on the use case, e.g. if the chunk is inhaled.

The assertion

```
acc(S[i].f, write)
```

is represented as chunk $\forall x \bullet x.\mathsf{f} \mapsto t_v \# count(t_{\mathsf{s}}[i..i+1], x)$. Again, the value $t_v$ depends on the use case. ⌟

### Axiomatization

We need to fulfil proof obligations over $count$: for example, if we want to read an index $i$ of sequence $S$, we will need to prove $count(S[0..|S|], S[i]) > 0$. Also, we need axioms to fulfil proof obligations that arise when an algorithm tries to split the permission to a sequence. Therefore, $count$ is axiomatised as follows:

$$\forall\, S \in \mathit{Seq}, x \in \mathit{Ref} \bullet count(S, x) \geq 0 \qquad \text{(non-negativity)}$$

$$\forall\, S \in \mathit{Seq}, x \in \mathit{Ref}, 0 \leq start \leq k \leq end \leq |S| \in Int \bullet \qquad \text{(splitting)}$$
$$count(S[start..end], x) = count(S[start..k], x) + count(S[k..end], x)$$

$$\forall\, S \in \mathit{Seq}, x \in \mathit{Ref}, 0 \leq start \leq i < end \leq |S| \in Int \bullet \qquad \text{(lookup)}$$
$$count(S[start..end], S[i]) > 0$$

Note that these axioms are not meant to be complete, but rather describe the operations that we want to support.

**Example 11.** Consider the partial specification of a mergesort in Listing 4.1. It takes a sequence, and two bounds. It is supposed to sort by recursively splitting the array, calling itself on the two parts. In the comments, the (simplified) state of the heap is shown. At the annotated position, the following proof obligation needs to be fulfilled with the axioms:

1. The call of sort requires the exhaling of permissions to

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \# count(S[start..\frac{start+end}{2}], x).$$

From the exhale algorithm, the proof obligation

$$\forall\, x \bullet 0 = count(S[start..\frac{start+end}{2}], x) \\ - \min(count(S[start..\frac{start+end}{2}], x), count(S[start..end], x))$$

arises, which can fulfilled by one application of the splitting axiom together with non-negativity.

Listing 4.1: A mergesort-like algorithm splits the sequence in the middle.

```
1  var f: Int
2
3  method sort(S:Seq[Ref], start:Int, end:Int)
4  requires 0 <= start && start <= end && end <= |S|
5  requires forall i:Int :: i in [start..end] ==> acc(S[i].f, write)
6  ensures forall i:Int :: i in [start..end] ==> acc(S[i].f, write)
7  {
8    // h : ∀x. x.f ↦ __ # count(S[start..end], x)
9    if(end-start>1) {
10     // (1)
11     // let the left part be sorted by a separate thread
12     exhale forall i:Int :: i in [start..(start+end)/2] ==> acc(S[i].f,
           write)
13     // h : ∀x. x.f ↦ __ # count(S[start..end], x) − count(S[start..(start + end)/2], x)
14     // let the right part be sorted by a separate thread
15     exhale forall i:Int :: i in [(start+end)/2..end] ==> acc(S[i].f, write)
16     // h : ∅
17     // both threads return
18     inhale forall i:Int :: i in [start..end] ==> acc(S[i].f, write)
19     // h : ∀x. x.f ↦ __ # count(S[start..end], x)
20     // merge...
21   }
22 }
```

⌟

## 4.4 Symbolic execution rules

Now, we can put together the functions defined above to modify the symbolic execution algorithm. We add two new rules for permissions under quantifiers, one for **produce** and one for **consume**. These are the rules that introduce quantified fields. We add four other rules to modify the behaviour of the symbolic execution *after* a field has been quantified.

### 4.4.1 Rules that introduce quantified chunks

The two rules illustrated in Figure 4.8 result in the introduction of quantified chunks for a specific field. As both rules begin identically, we start by introducing the identical part:

Both rules begin by temporarily adding the quantified variable with a fresh variable to the state. They then evaluate the condition to a term. They check if the condition is known to be false: if it is (e.g. `x in Set()`), then we can directly call the continuation: no access is obtained or taken away.

This check is also important because the rules assume the condition afterwards. If they assumed a term that introduced a contradiction, the following evaluations would always succeed, also if required permissions are not held. An example for this is the assertion `forall x:Ref :: x in Set() ==> acc(x.f, x.k)`, where the field `k` has type *Perm*. If we naively assumed $t_x \in Set()$, then any assertion after that could be proved, e.g. that we have access to `x.k`, although we might not have it.

Both rules then proceed to evaluate the receiver and the permission term. Also, they convert all chunks to quantified chunks using **quantifyChunksForField**. Then, the rules diverge:

- The first rule, the production rule, uses the **transform** function to get the quantified chunk. It then calls the continuation with the new chunk added to the heap. Additionally, it assumes that all receivers for which permission was given (i.e., for which $t2_\alpha > 0$) are non-null. This assumption is made because it is convenient: otherwise, each access predicate would have to be preceded by an additional assumption that the receiver is non-null.
- The second rule, the consumption rule, first gets the field value summary by calling **value**. This value has to be returned to the continuation as the snapshot[3]. Then, similarly to the production rule, a chunk is created by calling **transform**. Note that the value of this chunk is irrelevant, as **exhale** does never look at the value. That is why $null$ is used as the value. Function **exhale** returns the modified heap, which is passed to the continuation.

### 4.4.2 Modified rules for quantified fields

Once quantified chunks are on the heap for a certain field, additional rules modify the behaviour of old rules to make them compatible with quantified chunks. These rules apply only if **isQuantifiedForField** returns true, otherwise the symbolic execution falls back to the default rules. Figure 4.9 shows the modified rules for quantified fields. The following list describes the rules in the same order in which they appear in the figure:

- To evaluate a field access, we can make use of the previously defined function **value**. We defined it to additionally check if the required permissions are held. If calling **value** succeeds, the resulting term can be passed to the continuation.

[3] Snapshots are e.g. used to frame predicate instances, and orthogonal to this project.

$$
\begin{array}{l}
\textbf{produce'}(\sigma, tv, t_\beta, \texttt{forall x:T :: cond ==> acc(e.f, } \alpha\texttt{)}, Q) = \\
\quad \textbf{eval}(\sigma + (\mathsf{x}, \mathit{fresh}_{sort(\mathsf{T})}), \mathsf{cond}, (\lambda\ \sigma_2, t_{cond} \bullet \\
\quad\quad \textbf{if}\ \sigma_2 \vdash_{\mathsf{z3}} \neg(t_{\mathsf{cond}})\ \textbf{then}\ Q(\sigma_2.h, \sigma_2.\pi)\ \textbf{else} \\
\quad\quad\quad \textbf{eval}(\sigma_2 + t_{\mathsf{cond}}, \mathsf{e}, (\lambda\ \sigma_3, t_{\mathsf{e}} \bullet \\
\quad\quad\quad\quad \textbf{eval}(\sigma_3, \alpha, (\lambda\ \sigma_4, t_\alpha \bullet \\
\quad\quad\quad\quad\quad \textbf{let} \\
\quad\quad\quad\quad\quad\quad ch\ @\ \forall x \bullet x.\mathsf{f} \mapsto t_v\ \#\ t2_\alpha = \\
\quad\quad\quad\quad\quad\quad\quad \textbf{transform}(t_{\mathsf{e}}, \mathsf{f}, t_{\mathsf{cond}}, \mathit{fresh}_{\mathit{Ref} \to sort(\mathsf{f})}(t_{\mathsf{e}}), t_\alpha * t_\beta) \\
\quad\quad\quad\quad\quad\quad h = \textbf{quantifyChunksForField}(\sigma_4.h, \mathsf{f}) \\
\quad\quad\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad\quad\quad Q(h + ch, \sigma_4.\pi + (\forall x \in \mathit{Ref} \bullet t2_\alpha > 0 \implies x \neq null)))))))) \\
\\
\textbf{consume'}(\sigma, h, t_\beta, \texttt{forall x:T :: cond ==> acc(e.f, } \alpha\texttt{)}, Q) = \\
\quad \textbf{eval}(\sigma + (\mathsf{x}, \mathit{fresh}_{sort(\mathsf{T})}), \mathsf{cond}, (\lambda\ \sigma_2, t_{\mathsf{cond}} \bullet \\
\quad\quad \textbf{if}\ \sigma_2 \vdash_{\mathsf{z3}} \neg(t_{\mathsf{cond}})\ \textbf{then}\ Q(h, \sigma_2.\pi, unit)\ \textbf{else} \\
\quad\quad\quad \textbf{eval}(\sigma_2 + t_{\mathsf{cond}}, \mathsf{e}, (\lambda\ \sigma_3, t_{\mathsf{e}} \bullet \\
\quad\quad\quad\quad \textbf{eval}(\sigma_3, \alpha, (\lambda\ \sigma_4, t_\alpha \bullet \\
\quad\quad\quad\quad\quad \textbf{let}\ h_1 = \textbf{quantifyChunksForField}(h, \mathsf{f})\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{value}(\sigma_4, t_{\mathsf{e}}, \mathsf{f}, (\lambda\ \pi_1, t_{\mathsf{e.f}} \bullet \\
\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ ch = \textbf{transform}(t_{\mathsf{e}}, \mathsf{f}, t_{\mathsf{cond}}, null, t_\alpha * t_\beta)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{exhale}(\sigma_4, h_1, ch, (\lambda\ h_2 \bullet \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Q(h_2, \pi_1 - t_{\mathsf{cond}}, t_{\mathsf{e.f}})))))))))))
\end{array}
$$

Figure 4.8: Producing and consuming permissions under quantifiers introduces quantified chunks on the heap.

- When writing a field location, the write permission might be split across one or more quantified chunks. There are two options: first, we could update each value of the chunks that contain permission to the location. Second, we could exhale write permission to the written location and place a new chunk on the heap that contains the written value. We chose the latter, as it is straightforward to define with the given operations on quantified field chunks. We use the previously defined **transformElement** to get the suitable quantified chunk.
- When producing a (non-quantified) field access, we place the transformed chunk on the heap. Likewise, consuming a (non-quantified) field access for that field means transforming to a quantified chunk and exhaling that chunk.

## 4.5 Triggering strategy

It is challenging for a SMT solver to choose the terms with which to instantiate quantifiers. Instantiating with all possible combinations of terms that have been syntactically seen is infeasible. First of all, that number explodes quickly. Second, some quantifiers are prone to generate infinitely many instantiations. Consider an axiom for calculating the factorial

$$\forall\, n \in \mathit{Int} \bullet n > 0 \implies \mathrm{fac}(n) = n * \mathrm{fac}(n-1)$$

The prover might have seen a term $a + b$. So, if it instantiates the axiom naively, it will instantiate the axiom with $a + b$, which yields a new term $a + b - 1$. This new term can again be used to instantiate the axiom. This gives rise to a possibly infinite number of instantiations, a situation that is called *matching loop*[4].

This is why Z3 requires hints (so-called *triggers*) to control how and when to instantiate quantified expressions. A trigger is a set of expressions that the prover must have seen syntactically, modulo equality, to instantiate a quantifier. In particular, recursive functions require sensible choice of the trigger, as they produce syntactic expressions that could be used to instantiate the quantifier again. We want instantiations to be controlled by us, and therefore we have to provide sensible triggers.

We write a trigger in curly brackets before the body of the quantifier:

$$\forall\, x\ \{trigger\} \bullet body$$

### 4.5.1 Non-null checks

The symbolic execution rules assume the receiver of a field access is non-null when permissions are given. If the permissions are under quantifiers, this assumption is also a quantification. These quantifiers should be instantiated whenever the symbolic execution asserts receivers to be non-null, e.g. when a field is accessed. Therefore, we introduce a new function

[4]Note that the fact that the axiom is constrained to positive values does not affect the instantiation. Values are not taken into account when instantiating quantifiers.

$$
\begin{aligned}
&\textbf{eval'}(\sigma, \pi, \mathtt{e.f}, Q)\ \textbf{if isQuantifiedForField}(\sigma.h, \mathtt{f}) = \\
&\quad \textbf{eval'}(\sigma, \pi, \mathtt{e}, (\lambda\ \sigma_2, t_\mathtt{e} \bullet \\
&\qquad \textbf{value}(\sigma_2, t_\mathtt{e}.f, (\lambda\ \pi_1, t_{\mathtt{e.f}} \bullet \\
&\qquad\quad Q(\pi_1, t_{\mathtt{e.f}})))))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{exec}(\sigma, \mathtt{e.f\ :=\ g}, Q)\ \textbf{if isQuantifiedForField}(\sigma.h, \mathtt{f}) = \\
&\quad \textbf{eval}(\sigma, \mathtt{e}, (\lambda\ \sigma_2, t_\mathtt{e} \bullet \\
&\qquad \textbf{eval}(\sigma_2, \mathtt{g}, (\lambda\ \sigma_3, t_\mathtt{g} \bullet \\
&\qquad\quad \sigma_3 \vdash_{\mathtt{z3}} t_\mathtt{e} \neq \mathit{null} \wedge \sigma_3 \vdash_{\mathtt{z3}} \textbf{permission}(\sigma_3.h, t_\mathtt{e}.f) \geq 1 \wedge \\
&\qquad\quad \textbf{let}\ \mathit{ch} = \textbf{transformElement}(t_\mathtt{e}, \mathtt{f}, t_\mathtt{g}, 1)\ \textbf{in} \\
&\qquad\qquad \textbf{exhale}(\sigma_3, \mathit{ch}, (\lambda\ h' \bullet \\
&\qquad\qquad\quad Q(\sigma_3[h := h'] + \mathit{ch})))))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{produce'}(\sigma, tv, t_\beta, \mathtt{acc(e.f,\ }\alpha\mathtt{)}, Q)\ \textbf{if isQuantifiedForField}(\sigma.h, \mathtt{f}) = \\
&\quad \textbf{eval}(\sigma, \mathtt{e}, (\lambda\ \sigma_2, t_\mathtt{e} \bullet \\
&\qquad \textbf{eval}(\sigma_2, \alpha, (\lambda\ \sigma_3, t_\alpha \bullet \\
&\qquad\quad \textbf{let}\ \mathit{ch} = \textbf{transformElement}(t_\mathtt{e}, \mathtt{f}, \mathit{fresh}_{\mathit{sort}(\mathtt{f})}, t_\alpha * t_\beta)\ \textbf{in} \\
&\qquad\qquad Q(\sigma_3.h + \mathit{ch}, \sigma_3.\pi + (t_\mathtt{e} \neq \mathit{null})))))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{consume'}(\sigma, h, t_\beta, \mathtt{acc(e.f,\ }\alpha\mathtt{)}, Q)\ \textbf{if isQuantifiedForField}(\sigma.h, \mathtt{f}) = \\
&\quad \textbf{eval}(\sigma, \mathtt{e}, (\lambda\ \sigma_2, t_\mathtt{e} \bullet \\
&\qquad \textbf{eval}(\sigma_2, \alpha, (\lambda\ \sigma_3, t_\alpha \bullet \\
&\qquad\quad \textbf{value}(\sigma_3, t_\mathtt{e}.\mathtt{f}, (\lambda\ \pi_1, t_{\mathtt{e.f}} \bullet \\
&\qquad\qquad \textbf{let}\ \mathit{ch} = \textbf{transformElement}(t_\mathtt{e}, \mathtt{f}, \_, t_\alpha * t_\beta)\ \textbf{in} \\
&\qquad\qquad\quad \textbf{exhale}(\sigma_3, h, \mathit{ch}, (\lambda\ h_1 \bullet \\
&\qquad\qquad\qquad Q(h_1, \pi_1, t_{\mathtt{e.f}})))))))))
\end{aligned}
$$

Figure 4.9: If the heap is quantified for a field, slightly modified symbolic execution rules are used.

$$null : Ref \to Bool$$

and use $null(x)$ as a trigger for the non-null quantifiers:

$$\forall x \in Ref\,\{null(x)\}\ \bullet t_\alpha > 0 \implies x \neq null$$

The symbolic execution rules are adapted to assert $null(x) \vee x \neq null$ when asserting that a receiver is non-null, thus triggering the axioms.

**Example 12.** Producing the assertion `forall x:Ref :: x in S ==> acc(x.f, 1/2)` assumes $\forall\, x \in Ref\,\{null(x)\}\ \bullet \left((x \in t_S\ ?\ 1\ :\ 0) * \frac{1}{2} > 0\right) \implies x \neq null$. Now, when a location $t_y$.f with $t_y \in t_S$ is accessed, Silicon asserts $t_y \neq null \vee null(t_y)$. The term $null(t_y)$ triggers the quantifier and allows $t_y \neq null$ to be proved. ⌟

As the function $null$ is not exposed as SIL syntax, it cannot be used to trigger non-null assertions in a program. This results in an incompleteness described in subsection 5.3.3.

### 4.5.2 Value summaries

Field value summaries emit quantifiers to summarise heap contents. It is natural to choose the emitted function itself as a trigger. However, this is still incomplete:

**Example 13.** Consider Listing 4.2. It illustrates an incompleteness that would arise if only the fresh field value summary's function would be chosen as the trigger. The problem is that then the two summaries cannot be proved to be equal.

Listing 4.2: Choosing only the fresh field value summary's function as trigger is incomplete.

```
1  var f:Int
2  // ...
3  inhale forall x:Ref :: x in S ==> acc(x.f, 1/2)
4  //   h : ∀x • x.f ↦ t(x) # x ∈ t_S ? 1/2 : 0
5  inhale forall x:Ref :: x in S ==> x.f > 0
6  //   fresh t'
7  //   π : {∀x {t'(x)} • ((x ∈ t_S ? 1/2 : 0) > 0) ⟹ t'(x) = t(x), ∀x {} • x ∈ t_S ⟹ t'(x) > 0}
8  exhale forall x:Ref :: x in S ==> x.f > 0
9  //   fresh t''
10 //   π + ∀x {t''(x)} • ((x ∈ t_S ? 1/2 : 0) > 0) ⟹ t''(x) = t(x)
11 //   cannot prove assertion ∀x {} • x ∈ t_S ⟹ t''(x) > 0
12 //   as t''(x) = t'(x) cannot be established. The axiom for t' needs t(x) as a
       trigger.
```

⌟

That is why both sides of each equality axiom for a field value summary $t$ have to be chosen as the trigger: if the value of a chunk is a function application $t'(x)$, the equality axiom for that chunk has the form and triggers

$$\forall\, x \in Ref\,\{\{t(x)\}\,\{t'(x)\}\}\ \bullet t_\alpha > 0 \implies t(x) = t'(x)$$

### 4.5.3 Split axiomatisation

The splitting axiom is $count(S[start..end], x) = count(S[start..k], x) + count(S[k..end], x)$, which is prone to uncontrolled instantiation as it produces terms that are of the same shape. These terms not only trigger the splitting axiom again, but also other axioms of $count$ (e.g. the lookup axiom). Therefore, in the SMT solver, we encode this axiom differently: First, we introduce a new function

$$split : Seq \times Ref \times Int \to Bool$$

that carries no semantics, and exists only to trigger the splitting axiom. A term $split(S, x, i)$ indicates that $i$ is a valid index to split sequence $S$ when counting $x$. Splitting terms are generated from counting terms:

$$\begin{aligned}&\forall\, S \in Seq, x \in Ref, a, b \in Int\, \{count(S[a..b], x)\}\ \bullet \\ &\quad split(S, x, a) \land split(S, x, b)\end{aligned}$$

These terms trigger the splitting axiom rewritten with a new function $count' : Seq \times Ref \to Int$:

$$\begin{aligned}&\forall\, S \in Seq, x \in Ref, 0 \leq start \leq k \leq end \in Int \\ &\quad \{split(S, x, start), split(S, x, k), split(S, x, end)\}\ \bullet \\ &\qquad count'(S[start..end], x) = count'(S[start..k], x) + count'(S[k..end], x)\end{aligned}$$

We use $count'$ instead of $count$ to not produce new terms of $count$ that could be used to instantiate other axioms. We need additional axioms to extract meaning from the splitting axiom:

$$\begin{aligned}&\forall\, S \in Seq, x \in Ref\, \{count'(S, x)\}\ \bullet count'(S, x) \geq 0 \\ &\forall\, S \in Seq, x \in Ref\, \{count(S, x)\}\ \bullet count(S, x) = count'(S, x)\end{aligned}$$

### 4.5.4 Pure quantifiers and user-defined triggers

User-defined triggers are not yet supported.

# 5 Results

## 5.1 Applications

### 5.1.1 An access invariant for a union-find structure

A union-find structure manages the partition of a set. Each node points to a parent node, until eventually a node points to itself, which is the representant of its partition.

We model this structure as follows: One class, `UnionFind`, keeps the set of elements of the second class, `Node`, in the field `nodes`. Each of these nodes has a pointer `struct` back to its `UnionFind` structure. Listing 5.1 shows how an access invariant for this data structure can be defined. For an object of class `UnionFind`, predicate `inv` ensures:

- Write access for the set of nodes is held. Nodes can be added or removed.
- Read access for the `UnionFind` structure of each node is held. The idea is, that after a node is added, it is immutably bound to the structure. The invariant also makes sure that each node points correctly to its structure.
- Write access to the `parent` field of all nodes is held. This allows to modify the union-find structure itself, e.g. to implement `union`. Moreover, each parent is guaranteed to be non-null and in the set of nodes. Therefore, the structure can be traversed upwards while maintaining access.

Now, functions such as `allNodes` require only read access for `inv`, and can thus be executed in parallel. Methods such as `add` require `acc(inv(struct), write)` and thus have exclusive access to all nodes of the structure. Note that `add` needs to additionally ensure that `find` of all nodes except the added node has not changed. Let us take a close look at `add` and see where our algorithms are relevant. We omit the values of the chunks for simplicity.

Line 45 unfolds the invariant `inv`, which inhales its body. In particular, inhaling line 12 adds a quantified chunk (1) $\forall x \bullet x.\mathsf{parent} \mapsto \_ \# x \in t_{\mathsf{struct.nodes}} \; ? \; 1 \; : \; 0$. This causes chunks for field `parent` to be quantified, i.e. the chunk $t_{\mathsf{this}}.\mathsf{parent} \mapsto \_ \# 1$ (inhaled by line 34) is converted to a quantified chunk (2) $\forall x \bullet x.\mathsf{parent} \mapsto \_ \# x = t_{\mathsf{this}} \; ? \; 1 \; : \; 0$. Now, line 46 changes the set of all nodes to include the new node, i.e. we get a fresh term $t'_{\mathsf{struct.nodes}} = t_{\mathsf{struct.nodes}} \cup \{t_{\mathsf{this}}\}$. Line 47 folds `inv`, now line 12 has to be exhaled for the new set of nodes. This corresponds to a call to **exhale** that gets the required permissions from the chunks (1) and (2). It comes up with this interesting (simplified) proof obligation:

$$\forall y \bullet 0 = (y \in t'_{\mathsf{struct.nodes}} \; ? \; 1 \; : \; 0) - (y \in t_{\mathsf{struct.nodes}} \; ? \; 1 \; : \; 0) - (y \in \{t_{\mathsf{this}}\} \; ? \; 1 \; : \; 0)$$

It can be solved with the set axiomatisation.

Listing 5.1: A union-find structure can modelled with permission under quantifiers

```
// class UnionFind
var nodes: Set[Ref]
// class Node
var struct: Ref
var parent: Ref

predicate inv(this: Ref /* of class UnionFind */)
{
    acc(this.nodes, write)
    && forall o : Ref :: o in this.nodes ==> acc(o.struct, wildcard)
    && forall o : Ref :: o in this.nodes ==> o.struct == this
    && forall o : Ref :: o in this.nodes ==> acc(o.parent, write)
    && forall o : Ref :: o in this.nodes ==> (o.parent != null && (o.parent
        in (this.nodes)))
}

function allNodes(this:Ref /* of class UnionFind */): Set[Ref]
requires acc(inv(this), wildcard)
{
    unfolding acc(inv(this), wildcard) in this.nodes
}

function find(this:Ref /* of class Node */) : Ref
requires acc(this.struct, wildcard)
requires acc(inv(this.struct), wildcard)
requires valid(this)
{
    (unfolding acc(inv(this.struct), wildcard) in (this==(this.parent))) ?
        this : (unfolding acc(this.struct.inv(), wildcard) in find(this.
        parent))
}


/* add to structure */
method add(this: Ref, struct: Ref)
requires acc(inv(struct), write)
requires acc(this.parent, write)
 && ...
ensures ...
 && (this.struct == struct) && acc(inv(this.struct), write)
 && allNodes(struct) == old(allNodes(struct)) union Set(this)
 && this == find(this)
 && (forall o: Ref :: o in old(allNodes(struct)) ==> (unfolding acc(inv(
     struct), wildcard) in find(o)) == old(unfolding acc(inv(struct),
     wildcard) in find(o)))
{
    this.struct := struct
    this.parent := this

    unfold acc(inv(struct), write)
    struct.nodes := (struct.nodes union Set(this))
    fold acc(inv(struct), write)
}
```

Listing 5.2: Mergesort splits the sequence into two parts, an operation well supported by our axiomatisation.

```
// class Cell
var value:Int

// class ArrayOfInt
var array:Seq[Ref] // containing Cell(s)

method mergesort(a:Ref, b:Ref, start:Int, end:Int)
requires ...
 && forall i:Int :: i in [start..end] ==> acc(a.array[i].value, write)
 && forall i:Int :: i in [start..end] ==> acc(b.array[i].value, write)
ensures ...
 && forall i:Int :: i in [start..end] ==> acc(a.array[i].value, write)
 && forall i:Int :: i in [start..end] ==> acc(b.array[i].value, write)
 && forall i:Int :: i in [start..end-1] ==> b.array[i].value <= b.array[i
     +1].value
{
    var middle: Int
    if(end-start > 1)
    {
        middle := start + (end-start)\2
        mergesort(a,b, start,middle)
        mergesort(a,b, middle,end)
        merge(a,b,start,middle,end)
        // copy b to a...
    }
}

method merge(a: Ref, b:Ref, start:Int, middle:Int, end:Int)
requires ...
 && forall k:Int :: k in [start..end] ==> acc(a.array[k].value, write)
 && forall k:Int :: k in [start..middle-1] ==> a.array[k].value <= a.array[
     k+1].value
 && forall k:Int :: k in [middle..end-1] ==> a.array[k].value <= a.array[k
     +1].value
 && forall l:Int :: l in [start..end] ==> acc(b.array[l].value, write)
ensures ...
 && forall i:Int :: i in [start..end-1] ==> b.array[i].value <= b.array[i
     +1].value
{
    ...
}
```

### 5.1.2 Mergesort

Mergesort is among the first candidates to demonstrate our splitting of sequences. Listing 5.2 shows the specification. The method `mergesort` takes two arrays: `a` contains the values to be sorted, and `b` contains the result. To call it recursively, it also takes two bounds, `start` and `end`. It acquires permission to write the arrays `a` and `b` from `start` to `end`. In the

recursive case, it splits the array in the middle and calls `mergesort` two times. Each of these calls causes a call to **exhale**, which presents proof obligations similar to the ones in Example 11. After line 21, we have two chunks on the heap, one for the part of the sequence from start to mid, one from mid to end. Merge requires permission for all indices from start to end. So, the splitting axiom has to be applied backwards to combine the two chunks.

Method `merge` takes a partly sorted array and merges it. After exhaling its preconditions, the call to `merge` in line 22 then inhales its postcondition. Inter alia, this inhales a chunk (1) $\forall x \bullet x.\texttt{value} \mapsto t(x) \# count(t_{\texttt{b.array}}[start..end], x)$. It also inhales (2) $\forall i \bullet i \in [start..end - 1] \implies t(t_{\texttt{b.array}}[i]) \leq t(t_{\texttt{b.array}}[i+1])$ from line 34. This is simplified: in practice, both sides of the inequality would cause a new call to **value**, creating two fresh field value summary functions $t'$ and $t''$. The emitted axioms would then allow to show $t'(x) = t''(x)$. With the chunk (1) and axiom (2), it is possible to show lines 13 and 14 of mergesort's postconditions, respectively. Again, in line 14, each field access to `value` creates a fresh field value summary.

An orthogonal problem that is not yet reflected in this specification is to ensure that the sorted array is actually a permutation of the original array. It would be very easy to return a sequence $[1, 2, 3, \dots]$ and claim that this is the sorted sequence. Specifying permutations is challenging, but orthogonal to handling permissions under quantifiers.

### 5.1.3 Mutable Array

Listing 5.3 illustrates the specification of a (fixed-length) mutable array with permissions under quantifiers. Method `init` takes an uninitialised array, initialises it to a given size, and returns write permission for each index in the sequence. Note that it does not return write permissions for the array's underlying sequence itself, but only allows the caller to read the sequence. This effectively fixes the size and the locations of the array. Maybe the most subtle implication of `init` is that the locations of the array are mutually distinct: If for `i != j`, `this.array[i] == this.array[j]`, then more than write permission would be handed back for `this.array[i]` by init's postcondition.

Method `set` sets a value at a certain index of an array. Note that only write permissions for the specific index are required, which allows to conclude that the other values are left unchanged. `get` is implemented as a pure function.

Listing 5.3: A mutable array can be specified with permissions under quantifiers.

```
1  var value: Int      /* class Location */
2  var array: Seq[Ref] /* class MutableArray */
3
4  method   init(this:Ref, size:Int)
5  requires acc(this.array, write)
6  ensures  acc(this.array, wildcard)
7  ensures  |this.array| == size
8  ensures  forall i:Int :: i in [0..size] ==> acc(this.array[i].value, write)
9  {
10     assume |this.array| == size
```

```
11      /* generate new permissions for the array */
12      inhale forall i:Int :: i in [0..size] ==> acc(this.array[i].value,
            write)
13  }
14
15  method   set(this:Ref, index:Int, val:Int)
16  requires acc(this.array, wildcard)
17  requires 0 <= index && index < |this.array|
18  requires acc(this.array[index].value, write)
19  ensures  acc(this.array, wildcard)
20  ensures  acc(this.array[index].value, write)
21  ensures  this.array[index].value == val
22  {
23      this.array[index].value := val
24  }
25
26  function get(this:Ref, index:Int):Int
27  requires acc(this.array, wildcard) &&
28  requires 0 <= i && i < |this.array|
29  requires acc(this.array[index].value, wildcard)
30  {
31      this.array[index].value
32  }
```

## 5.2 Runtime performance for programs without permissions under quantifiers

Our changes to Silicon influence the performance also for programs without permissions under quantifiers. We do not use quantified chunks for these, as they require more calls to Z3. This keeps the performance loss due to quantified chunks minimal.

Table 5.1 compares Silicon with and without quantified chunks. It shows the test cases with the highest runtime increase.

The following changes of the original code cause the performance drop:

- When proving that a receiver is not null, Silicon uses the null trigger (see subsection 4.5.1), also if there are no quantified chunks on the heap. The consequence is that some "shortcut" decisions of Silicon, e.g. if an assertion is syntactically contained in the path conditions, no longer apply. Thus, Silicon calls Z3 slightly more often.
- We added additional axioms for sequences. They lead to more quantifier instantiations when sequences are used, even in a context unrelated to permissions.
- The symbolic execution checks **isQuantifiedForField** to choose between the rule for quantified chunks and the rule for traditional chunks. These checks are already cheap, but they could be improved by keeping a flag for each field.

Table 5.1: Comparison of the runtime performance for programs without permissions under quantifiers. The table shows the ten programs for which the runtime increases most.

| Test case | Quantifier instantiations | Time |
|---|---|---|
| issues/carbon/0002.sil | ±0 | +21ms (20%) |
| chalice/test8.sil | ±0 | +56ms (14%) |
| sequences/nil.sil | ±0 | +28ms (14%) |
| issues/silicon/0053.sil | ±0 | +39ms (14%) |
| basic/unique.sil | ±0 | +29ms (13%) |
| basic/func3.sil | ±0 | +37ms (13%) |
| issues/silicon/0039b.sil | ±0 | +71ms (12%) |
| sequences/sequences.sil | +245 (36%) | +33ms (12%) |
| issues/sil/0008.sil | ±0 | +33ms (12%) |
| chalice/AVLTree.nokeys.sil | +118 (1%) | +4234ms (11%) |

## 5.3 Completeness

A number of incompletenesses are induced by our handling of permissions under quantifiers and have not been not solved at the time of writing. We distinguish between incompletenesses in supported idioms, and unsupported idioms, where the latter are discussed in section 5.5.

### 5.3.1 Multiple indices point to the same object

For sequences, multiple indices can point to the same object. This influences exhaling: Listing 5.4 shows a situation where it should be possible to exhale write permissions for an object that is known to be at index $0$ and $1$ of a sequence $S$, each giving half permission to a field.

- Mentioning a sequence at a certain index does not trigger the splitting axiom, i.e. it is unknown that $count(S[0..2], S[0]) = count(S[0..1], S[0]) + count(S[1..2], S[0])$. Such a split would currently only be triggered by a field write. Triggering the split with a field access is possible, but presumably influences performance, as more quantifier instantiations are triggered.
- There are no base cases for $count$. The lookup axioms yields $count(S[0..1], S[0]) \geq 1$, but nothing is known for $count(S[1..2], S[0])$. It seems that adding an axiom for the base case

  $$\forall\, S \in Seq, x \in Ref, a \in Int\ \ \{count(S[a..a+1], x)\} \bullet \\ a \geq 0 \wedge a < |S| \wedge S[a] = x \implies count(S[a..a+1], x) = 1$$

  would suffice.

Listing 5.4: The axiomatisation does not allow exhales over multiple indices of a sequence.

```
1  var f:Int
2
3  method m(S:Seq[Ref])
4  requires |S| == 2
5  requires S[0] == S[1]
6  requires forall i:Int :: i in [0..2) ==> acc(S[i].f, 1/2)
7  // fails: Insufficient permission to S[0].f
8  ensures acc(S[0].f, write)
9  {
10 }
```

### 5.3.2 It is not possible to mix idioms

The current axiomatisation of sets and sequences reaches its limits when sets and sequences are mixed. In general, when more idioms are added, it has to be decided which idioms should be able to play well together. For example, we did not focus on mixing sets and sequences as we had no real-world use case, but that may change.

Listing 5.5 illustrates this incompleteness. It fails with the message that y could be null in the postcondition. This is because the axiomatisation fails to show that for $t_y \in Set\,(t_x)$, it holds that $count\,(Seq\,(t_x)\,[0..|Seq\,(t_x)\,|], t_y) > 0$.

Fixing this special case might be easy by adding a special axiom. Nevertheless, it may be harder to come up with a small set of general axioms that allow sets and sequences to be mixed.

Listing 5.5: Sets and sequences cannot be mixed.

```
1  var f:Int
2
3  method m(S1:Seq[Ref],S2:Set[Ref], x:Ref)
4  requires S1 == Seq(x)
5  requires forall i:Int :: i in [0..|S1|) ==> acc(S1[i].f, write)
6  requires S2 == Set(x)
7  // fails: "y could be null"
8  ensures forall y:Ref :: y in S2 ==> acc(y.f, write)
9  {
10
11 }
```

### 5.3.3 Non-nullness cannot be asserted

In subsection 4.5.1, we introduced a function $null$ to trigger the non-null assumptions. This works fine when Silicon proves that a receiver is non-null, e.g. when accessing a field. However, the assumption is not triggered if the specification manually asserts that an object is not null, such as in Listing 5.6.

It would be ideal to trigger the non-null assumptions when seeing $\neq null$ as a term. Unfortunately, "=" is a special operator for Z3 and cannot be used as a trigger.

- A simple solution would be to syntactically check for non-null assertions and emit the trigger. But, this would fail whenever $\neq null$ is not seen syntactically, e.g. only modulo equality.
- One could introduce a special referential equality function $Eq(x, y)$ and use this as a trigger.
- Instead of having $null$ as an object literal, one could attach semantics to the function $null(x)$.

Listing 5.6: Manual assertions that objects are non-null fail.

```
1  var f:Int
2
3  method m(S:Set[Ref], b:Ref)
4  requires forall s:Ref :: s in S ==> acc(s.f, write)
5  requires b in S
6  {
7    // works: Silicon assumes the nullTrigger
8    var t:Int := b.f
9    // fails: "b != null might not hold"
10   assert(b != null)
11 }
```

### 5.3.4 Distinctness cannot be asserted

It is not possible to manually assert the distinctness of objects when they are distinct because write permission was inhaled to each of them (see Listing 5.7). Logically, one can infer `x != y` from `acc(x.f, 1) && acc(y.f, 1)`. In practice, this knowledge is not always available, because the heap is kept separate from the prover. However, it is available implicitly: as shown in Listing 5.7, it can be proved that `S[b].f` has not changed after writing `S[a].f`.

To establish distinctness explicitly, one would have to use the fact that no more than full permission can ever be distributed for each field location. This is captured by the "axiom"

$$\forall\, h, x, \mathtt{f} \bullet \textbf{permission}(h, x, \mathtt{f}) \leq 1$$

This cannot be a real axiom encoded in Z3, as **permission** is a function in the verifier. Instead, it has to be emitted for a specific heap $h$, receiver $x$, and field name `f` by computing the term $t_r := \textbf{permission}(h, x, \mathtt{f})$ in the verifier and then emitting $t_r \leq 1$. Therefore, one has to decide when, and for which location to emit this axiom. One could, for example, emit this axiom every time **permission** is called. In our implementation, we chose not to emit it at all, as it affects completeness only if distinctness is asserted explicitly, which was not necessary for our examples.

**Example 14.** If the chunks

$$\forall x \bullet x.\mathtt{f} \mapsto \_ \# x = t_x \; ? \; 1 \; : \; 0 \qquad\qquad \forall x \bullet x.\mathtt{f} \mapsto \_ \# x = t_y \; ? \; 1 \; : \; 0$$

are on the heap, **permission**$(h, t_x, \mathtt{f})$ results in the term $(t_x = t_x \; ? \; 1 \; : \; 0) + (t_x = t_y \; ? \; 1 \; : \; 0)$, which can be simplified to $t_r := 1 + (t_x = t_y \; ? \; 1 \; : \; 0)$. If we assume $t_r \leq 1$, we can deduce $t_x \neq t_y$. ⌟

Listing 5.7: Distinctness cannot be explicitly asserted.

```
1  var f:Int
2
3  method m(S:Seq[Ref], a:Int, b:Int)
4  requires forall i:Int :: i in [0..|S|) ==> acc(S[i].f, write)
5  requires a >= 0 && a < b && b < |S|
6  {
7    S[a].f := 5
8    // works
9    assert(S[b].f == old(S[b].f))
10   // fails: S[a] != S[b] might not hold
11   assert(S[a] != S[b])
12 }
```

## 5.4 Known Issues

### 5.4.1 Sort wrappers for functions

Silicon uses snapshots to summarise the relevant state of the heap when folding a predicate. To this end, every result of **consume** has an associated snapshot, which is returned as the second parameter to the continuation of **consume**.

For simple chunks, this snapshot is just the value, converted to a snapshot by a so-called sort wrapper. For quantified chunks, the snapshot is still the value. However, there are currently no sort wrappers for functions.

This results in an incompleteness illustrated in Listing 5.8. Values in quantified chunks are not preserved when folding and unfolding predicates.

This issue of missing sort wrappers is orthogonal to the symbolic execution rules, except for a minor detail: the production rule in Figure 4.8 should use snapshot $tv$ instead of *fresh* as the value for the new chunk.

Another, more complete solution may be to introduce sort wrappers for functions. However, we did not look further into this, as Silicon's way of producing fresh snapshots is likely to change in the future[1].

[1] Personal communication with Malte Schwerhoff, 12.2.2014

Listing 5.8: It cannot be proved that the value of a quantified chunk is preserved after folding and unfolding.

```
1  var f:Int
2
3  predicate inv(s:Set[Ref]) {
4          (forall x : Ref :: x in s ==> acc(x.f, write))
5  }
6
7  method m1(this:Set[Ref])
8  requires acc(inv(this), write)
9  ensures acc(inv(this), write)
10 {
11         unfold acc(inv(this), write)
12         var t:Ref
13         assume (t in this)
14         var a:Int := t.f
15         // no sort wrapper for functions, value of chunk is stored "
               directly" in snapshot
16         fold acc(inv(this), write)
17         unfold acc(inv(this), write)
18         // fails because the quantified chunk has a fresh value instead
               from the snapshot of inv
19         assert a == t.f
20
21 }
```

### 5.4.2 The composite pattern

The composite pattern is a design pattern and was proposed as a verification challenge in [LLM07]. Smans et al. verified it using Verifast and predicates [JSP08]. It would be very interesting to verify it using permissions under quantifiers, as we could then directly compare the specifications. However, our implementation does not terminate for our specification.

Consider Listing 5.9, which in particular shows our attempt to specify an invariant for the composite in predicate `inv`. Upon closer inspection, we see that `inv` is supposed to hold the access to the set of all nodes. Each node has two pointers, `left` and `right`, to its subnodes. To let us traverse the structure, line 18 and 21 specify inclusion properties. Both have the form `forall x:Ref :: in(x) && ... ==> in(left(x)) && ...`.

This form is prone to a matching loop: Z3 might choose the naive trigger `in(x)`. Then, the axiom can be instantiated with `in(left(x))`, `in(left(left(x)))`, .... As the symbolic execution does not terminate, apparently Z3 does choose such a trigger[2].

This suggests that the problem is caused by the lack of good triggers for these inclusion properties. We do not support triggers for user-supplied quantifiers yet. Solving this problem is orthogonal to the handling of permissions.

[2]Note that line 13 of our union-find example Listing 5.1 has the same form, but union-find does terminate. We suspect that in this simpler case, Z3's heuristics detect the matching loop.

Listing 5.9: The invariant predicate emits quantifiers that cause our implementation not to terminate.

```
// class "Node"
var value: Int
var left: Ref
var right: Ref
var parent: Ref
var parentStruct: Ref /* pointer to the node's composite */

// class "Composite"
var all: Set[Ref]


predicate inv(this: Ref)
{
     acc(this.all, write)
     && forall m: Ref :: m in this.all ==> acc(m.parent, wildcard)
     && forall o: Ref :: o in this.all && o.parent!= null ==> acc(o.
         parentStruct, wildcard)
     && forall q: Ref :: q in this.all ==> acc(q.left, 1/2)
     && forall r: Ref :: r in this.all && r.left != null ==> r.left in this
         .all && t.left.parent==this
     && forall s: Ref :: s in this.all && s.left == null ==> acc(s.left,
         1/2)
     && forall v: Ref :: v in this.all ==> acc(v.right, 1/2)
     && forall w: Ref :: w in this.all && w.right != null ==> w.right in
         this.all && x.right.parent==this
     && forall y: Ref :: y in this.all && y.right == null ==> acc(y.right,
         1/2)
     && forall x: Ref :: x in this.all ==> acc(x.value, write)
     && forall y: Ref :: y in this.all ==> y.value == (y.left==null ? 0 : y
         .left.value) + (y.right==null ? 0 : y.right.value)  + 1
}

function valid(this: Ref) : Bool
  requires acc(this.parentStruct, wildcard) && this.parentStruct!=null &&
      acc(inv(this.parentStruct), wildcard)
{
  unfolding acc(inv(this.parentStruct), wildcard) in (this in (this.
      parentStruct.all))
}

function getLeft(this: Ref) : Ref
  requires acc(this.parentStruct, wildcard) && this.parentStruct!=null &&
      acc(inv(this.parentStruct), wildcard) && valid(this)
{
  unfolding acc(inv(this.parentStruct), wildcard) in this.left
}
```

## 5.5 Unsupported Idioms

### 5.5.1 Multiple field access

A new idiom could have the form `forall y:Ref :: y in S ==> acc(y.f.g, p)`, where `f` is a field of type `Ref`. An object `y.f` is then called a *pivot*. For an object `x`, we then have `p` permission for each `y in S` with `y.f = x`.

It is unclear how to count these elements. A simple, but incomplete solution could be to transform to the chunk $\forall x \bullet x.\texttt{f} \mapsto \_ \ \# \ (\exists \ y \bullet y \in t_S \wedge t_y(y) = x) \ ? \ t_p \ : \ 0$. This does not regard that two elements might point at $x$, inhaling $2 * t_p$ permission. It is also unclear to us how an algorithm would use this idiom.

### 5.5.2 Additional constraints for sequences

Currently, we are very restrictive about the syntax for sequences. It could be that some algorithms want to split the sequence differently, e.g. in odd and even indices (see Listing 5.10).

Presumably, such requirements could be captured in special axioms for *count*.

Listing 5.10: An algorithm could split a sequence in even and odd indices.

```
1 var f:Int
2 method even_odd(S:Seq[Ref])
3 requires forall i:Int :: i in [0..|S|] ==> acc(S[i].f, write)
4 {
5     exhale forall i:Int :: i in [0..|S|] && i % 2 == 0 ==> acc(S[i].f,
          write)
6     exhale forall i:Int :: i in [0..|S|] && i % 2 == 1 ==> acc(S[i].f,
          write)
7 }
```

### 5.5.3 Multiple quantifiers

We do not support more than one quantified variable. Two quantified variables could be useful e.g. to give access to elements of a multidimensional sequence. This is illustrated in Listing 5.11. Note that the syntax given in the example does not exist in SIL yet.

Listing 5.11: Two quantified variables could be used to give access to a matrix-like structure.

```
1 var f:Int
2 method multiply(A:Seq[Ref][Ref],B:Seq[Ref][Ref])
3 requires forall i,j:Int :: i in [0..|A|] && j in [0..|A|] ==> acc(S[i][j].f
      ,write)
4 {}
```

# 6 Conclusion

We extended Silicon to support permissions under quantifiers for sets and sequences. To this end, we introduced a new kind of heap chunk called quantified field chunk. In contrast to the old field chunks, quantified chunks allow the representation of access to many field locations on the heap. We defined the permissions held for a receiver, exhaling, and value lookup for quantified chunks and gave the modified symbolic execution rules. Quantified chunks are only introduced when a program uses permissions under quantifiers.

With permission under quantifiers, we showed how to model the following data structures and algorithms: a union-find structure can be specified by a predicate that keeps access to the set of all nodes, where each node's parent is specified to be in this set. A parallel mergesort can be specified by splitting the array in two parts, an operation that is well supported by our axiomatisation. Also, we showed how to model a mutable array with a predicate that keeps access to all indices of a sequence.

Our approach is limited by the axiomatisation for sets and sequences. Access to sequences has to be specified in explicit subranges, e.g. $S[a..b]$ for a sequence $S$ and integers $a$, $b$ with $0 \leq a \leq b \leq |S|$. We do not support multiple field access, access to sequences that cannot be specified in explicit subranges (e.g., splitting in even and odd indices), and multiple quantifiers (e.g., to access matrices).

Let us reflect on our initial goal: to elegantly specify algorithms on unbounded data with sets and sequences. Quantified permissions are a solution for this problem: they specify access to sets and sequences without dictating a way of traversal. We showed how permissions under quantifiers can be implemented in Silicon with quantified chunks, and demonstrated the usefulness of our solution.

## 6.1 Related work

Permissions under quantifiers are closely related to the iterated star, a commonly used idiom in separation logic [Rey02].

Smans et al. implemented the iterated star in Vericool [SJP09]. Their assertions have the form $\forall^* x \in (min \ : \ max) \bullet \phi$, where $min$ and $max$ are integer expressions. Smans et al. give the intuition that the latter assertion states that $\phi$ holds for all integers between $min$ (inclusive) and $max$ (exclusive), and that for any two different integers in that range, the locations required to be accessible by $\phi$ are *disjoint*. This is different to our fractional permissions under quantifiers. For example, when specifying access to a sequence, the objects at two different indices do not have to be disjoint. In addition, our approach also

supports sets, which does not seem to be expressible in the form given by Smans. To our knowledge, the iterated star is only supported in the VCG variant of VeriCool. This makes the implementation different.

As far as we know, no other automated verifier supports permissions under quantifiers.

## 6.2 Future work

### 6.2.1 Native support for arrays

SIL and Silicon do not natively support arrays yet. In our work, we mostly used sequences to model arrays. Arrays would therefore be a good addition to the language. We showed how to model arrays with sequences and permission under quantifiers. This could be used to implement arrays natively by keeping a dedicated field to hold the array values, alongside a sequence.

### 6.2.2 General axiomatisation for counting

It would be interesting to know whether there is a general solution for counting (see subsection 4.3.2), and if it can be encoded as a formula for a SMT solver. Such a general approach would solve our issue of unsupported idioms and e.g. allow the specification of permissions for multiple field accesses. We suspect that this is not possible with our approach, but made no efforts to support this claim.

### 6.2.3 Analyse usefulness of quantified chunks for other applications

While implementing quantified chunks, we noticed that they can also be useful in situations unrelated to permissions under quantifiers. Consider Listing 6.1, which fails to verify, also in Silicon with quantified chunks. The reason is that it does not use permissions under quantifiers, and hence the field `f` is never quantified. However, as laid out in the comments, it could be solved with quantified chunks and the operations on them. The example is artificial, and we are unsure whether this would be needed in practice. It could, however, be worthwhile to implement a backtracking algorithm that uses quantified chunks when the verification fails due to insufficient permission.

Listing 6.1: Quantified chunks could solve incompletenesses that do not regard permissions under quantifiers.

```
1  var f:Int
2
3  method q(a:Ref, b:Ref, c:Ref)
4  requires acc(a.f, write) && acc(b.f, write)
5  requires c == a || c == b
6  {
7    // fails with simple chunks, as neither c==a nor c==b can be proved
```

```
8     c.f := 5
9     // it would, however, work with chunks
10    // ∀x • x.f ↦ _ # x = t_a ? 1 : 0
11    // ∀x • x.f ↦ _ # x = t_b ? 1 : 0
12    // because then it can be proved that full permission to c.f is held
13  }
```

# Bibliography


[Boy03] John Boyland. Checking interference with fractional permissions. In *Static Analysis*, pages 55–72. Springer, 2003.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DP08] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. *ACM Sigplan Notices*, 43(10):213–226, 2008.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of programming languages*. MIT press, 2001.

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW520, Department of Computer Science, K.U.Leuven, 2008.

[JSP08] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, page 83, 2008.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[LM09] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. *Programming Languages and Systems*, pages 1–16, 2009.

[LMS09] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[Sch11] Malte Schwerhoff. Symbolic Execution for Chalice. Master's thesis, ETH Zurich, 2011.

[SD13] Alexander J Summers and Sophia Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. *ECOOP 2013–Object-Oriented Programming*, page 129, 2013.

[SJP08] Jan Smans, Bart Jacobs, and Frank Piessens. VeriCool: An automatic verifier for

a concurrent object-oriented language. In *Formal Methods for Open Object-Based Distributed Systems*, pages 220–239. Springer, 2008.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009–Object-Oriented Programming*, pages 148–172. Springer, 2009.

[SJP10] Jan Smans, Bart Jacobs, and Frank Piessens. Heap-dependent expressions in separation logic. In *Formal Techniques for Distributed Systems*, pages 170–185. Springer, 2010.