Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Enhancing Deductive Verification of Unbounded Heap Data Structures

Master's Thesis

Markus Limbeck

March 26, 2025

Advisors: Dr. Marco Eilers, Dionysios Spiliopoulos, and Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

Formal verification ensures software correctness beyond traditional testing by using mathematical proofs to establish program properties. Viper is an automated, permission-based verification infrastructure that uses Implicit Dynamic Frames, a variation of separation logic, to reason about heap properties. To handle unbounded heap data structures, Viper employs two expressive concepts: iso-recursive predicates and iterated separating conjunctions (ISCs). While powerful, these constructs are complex to reason about automatically and can negatively affect verification performance.

This work investigates performance challenges in Viper, identifying key inefficiencies and proposing optimizations. Contributions include limiting predicate permissions to simplify verification across both of Viper's backends, introducing a greedy approach to ISCs that reduces reliance on heuristics, and extending state consolidation with multiple strategies for Viper's symbolic execution backend. Further optimizations focus on reducing quantified axioms and quantifier instantiations, easing proof obligations for the underlying SMT solver. Additionally, a novel mechanism for reusing inverse functions used in Viper's SMT encoding eliminates redundant axioms, further improving efficiency.

All contributions and optimizations are fully implemented and evaluated with respect to their impact on quantifier instantiations, verification time, and other relevant metrics. The only exception is the concept of reusing inverse functions, which has been explored only through preliminary testing and has not yet been fully implemented.

Together, these optimizations enhance Viper's ability to verify unbounded data structures.

Acknowledgements

First and foremost, I would like to thank my supervisors, Dr. Marco Eilers and Dionysios Spiliopoulos, for their outstanding support throughout my work on this thesis. Their guidance during our weekly meetings and their helpful advice whenever I faced challenges were invaluable.

I would also like to extend my sincere thanks to Prof. Dr. Peter Müller and the entire Programming Methodology Group for giving me the opportunity to contribute to their research and ongoing projects.

Lastly, I would like to thank my parents for always believing in me and supporting me throughout this journey. This study would not have been possible without their encouragement and generosity.

Contents

Chapter 1

Introduction

Writing bug-free software is inherently challenging. Traditional testing methods cannot fully guarantee the absence of errors. With this in mind, a more robust approach is the use of formal methods which leverage rigorous mathematical proofs to establish correctness properties. This process is known as program verification.

The underlying logic that defines the verification techniques used in this work is called implicit dynamic frames [19], a variation of separation logic [17] which provides a formal theory to reason about mutable state and aliasing. In order to express properties involving unbounded heap data structures, implicit dynamic frames provides two key concepts: iso-recursive predicates and iterated separating conjunctions (ISCs) [15]. Iso-recursive predicates are used to model recursive data structures, such as linked lists. In contrast, ISCs support a more flexible representation, enabling the modeling of random access data structures like arrays.

A verification infrastructure is a framework that, given the source code of a program and its specification, provides the necessary tools to automatically check whether the source code adheres to its specification. This work focuses on Viper [14], short for Verification Infrastructure for Permission-based Reasoning. Viper provides an intermediate language, also named Viper. Furthermore, it offers a variety of frontends, such as Gobra [3] for Go, which automatically translate programs annotated with custom specifications into Viper's intermediate language.

To verify a program, Viper leverages SMT solvers like Z3 [13] that can efficiently check satisfiability in logical constraint. These constraints are derived by translating the program using either its symbolic execution (SE) backend or its verification condition generation (VCG) backend. Within the Viper infrastructure, these are referred to as Silicon [18] and Carbon [1], respectively.

In this work, we investigate Viper's performance issues related to ISCs and iso-recursive predicates, identifying performance bottlenecks and exploring potential optimizations. The key challenge is to balance completeness and performance in a controlled manner, ensuring that incompleteness does not impact the verification of a large set of real-world programs.

The main contributions can be summarized as follows.

- **Limiting predicate permissions:** A new feature that restricts the amount of permissions of a predicate can be held, simplifying predicate usage for both the user and the SMT solver
- **Greedy ISC approach:** A less heuristic-dependent algorithm for reasoning about ISCs in Viper's SE backend, leading to more stable verification in practice.
- **State consolidation for ISCs:** An extension of state consolidation for ISCs in Viper's SE backend, incorporating four distinct heuristics to improve completeness of the new greedy approach.
- **General optimizations for ISCs:** Several improvements aimed at reducing the number of quantified axioms and subsequent quantified instantiations, thus simplifying the proof obligations for the SMT solvers.
- **Reusing user-defined inverse functions:** A novel concept that reuses user-provided axioms to define inverse functions, which are internally used by Viper when reasoning about ISCs, rather than generating new ones.

This thesis is organized into six chapters. The second chapter lays out the background and core concepts used throughout the work. The third chapter introduces a new predicate feature. The fourth chapter presents the new greedy approach for ISCs, along with the state consolidation extension. The fifth chapter explores various optimizations aimed at reducing definitional axioms and minimizing quantified instantiations, including the reuse of user-defined inverse functions. Finally, the sixth chapter concludes the thesis and discusses directions for future work.

Chapter 2

Background

In this section, the key concepts essential for the following chapters are introduced. It begins with general techniques related to program verification and concludes with specifics about the verification infrastructure Viper.

2.1 Program verification

Program verification is the process of proving that a program adheres to formal specifications. Usually, this is achieved by translating a program into a logical formula, whose validity implies correctness with respect to the given specifications. Such validity is checked by a satisfiability modulo theories (SMT) solver like Z3 [13].

The validity check requires negating the proof obligation and proving that the resulting formula is unsatisfiable. This ensures that all possible paths where the proof obligation does not hold lead to contradictions, meaning they are impossible to reach. Consequently, it is sound to conclude that the original proof obligation is valid. However, this approach carries a risk. This is only sound if the original formula is not a contradiction by itself.

2.1.1 Program correctness

In program verification, correctness with respect to the given specifications is formally expressed using Hoare triples [10], which provide a structured way to define specifications using logical formulas.

A Hoare triple takes the form {P} S {Q}, where P (the precondition) must hold before executing the statement S, and after executing S, Q (the postcondition) is guaranteed to hold if P was initially true. This concept is used to prove that certain conditions hold before and after execution of statements. Combining all Hoare triples of all program's statements allows reasoning

about the program's overall correctness. This combining is accomplished using either verification condition generation or symbolic execution.

2.1.2 Permission-based reasoning

Imperative programming languages are based on program state modification, making it essential to reason about these state changes. Permission-based reasoning introduces a formal notion of permissions to manage access to memory locations and reason about how state modifications affect their values.

Approaches such as implicit dynamic frames [19], a variation of separation logic [17], leverage permissions to ensure that certain parts of the heap remain unaffected by a given operation, a concept formally known as framing.

This concept extends to fractional permissions [7], which differentiate between full permissions, granting write access, and partial permission, allowing only read access. Importantly, holding any positive amount of permission to a memory location guarantees that its value remains unchanged.

2.1.3 Verification condition generation

Verification condition generators (VCGs) use predicate transformers, such as weakest precondition (WP) transformer, to compute a verification condition (VC). The validity of this VC ensures a program's correctness against its specification and can be checked by SMT solvers.

For imperative languages that modify a program's state, heap can be modeled as a global mapping from locations to values and passed to the solver along with the VC. This enables reasoning about heap properties like aliasing.

A key feature of this approach is that only one VC per method is generated. The advantage is that the SMT solver has all relevant information and can optimize accordingly. However, the downside is that the VC is usually very large.

2.1.4 Symbolic execution

Symbolic execution (SE) verifies programs by executing them symbolically, using symbolic values rather than concrete ones. During execution, it maintains path conditions which track constraints on symbolic values.

When the control flow branches, SE explores all possible execution paths, updating the path condition by conjoining it with the corresponding branch condition. Assertions are then verified using an SMT solver which checks their validity under the current path condition.

A drawback of this approach stems from exponential growth of execution paths. Additionally, branches are likely to contain redundant proof obligations, making it inefficient as work is unnecessarily repeated.

However, this also allows for handling heap properties within the verifier instead of encoding them in proof obligations. While this simplifies proof obligations it introduces incompleteness, as heap-related information is not always shared with the prover.

SE represents heap locations using heap chunks of the form $[r.f \rightarrow v]$, where r is a reference, f is a field that, together with r, identifies the heap location, and v is a symbolic value representing the value stored at that location.

Heap chunks are stored in a multiset called the symbolic heap. The verifier can then determine whether permissions are available for a specific location either through syntactic equality or by leveraging aliasing information from the path conditions via the SMT solver.

For example, given a reference to a location $r_2.f$, a path condition $\{r_1 = r_2\}$, and a symbolic heap $\{\{[r_1.f \rightarrow v]\}\}$, the SMT solver can prove that r_2 and r_1 are aliases. Based on this, the verifier can conclude that permissions for $r_2.f$ are available.

This concept can also be extended to fractional permissions, as described in Section 2.1.2, meaning a heap chunk takes the form $[r.f \rightarrow v \# p]$, where $\mathtt{p}$ denotes a fractional permission amount. However, this also introduces two sources of incompleteness.

First, the verifier may underestimate available permission amounts, without using path conditions and the SMT solver. For example, if the symbolic heap contains the chunks $[r_1.f \rightarrow v_1 \# 0.5]$ and $[r_2.f \rightarrow v_2 \# 0.5]$, and the path condition implies $r_1 = r_2$, the verifier cannot infer full (write) permission for $r_1.f$ with only syntactic equality.

Second, the SMT solver may fail to establish that $r_1 = r_2$ implies $v_1 = v_2$, because this implication depends on the symbolic heap. A common approach to mitigate this problem is state consolidation. This technique leverages information from the path conditions to prove aliasing between references and subsequently merges the corresponding chunks into a single chunk.

In the case of $r_1 = r_2$, state consolidation removes the two chunks and replaces them with $[r_3.f \rightarrow v_3 \# 1]$. Additionally, new path conditions are added to ensure equality: $\{r_1 = r_3, r_2 = r_3, v_1 = v_3, v_2 = v_3\}$.

2.1.5 Quantified formulas

Handling quantified formulas is a challenging problem. However, since only the unsatisfiability of the negated formula needs to be proven, a simple yet

effective technique can be used. This technique, known as e-matching [9], works by dividing the formula into ground terms (quantifier-free parts) and quantified parts. Each quantifier is assigned a trigger which, when it syntactically appears in a ground term, instantiates the quantifier with the corresponding term modulo equality. With well-chosen triggers, this approach performs well in practice.

However, completeness is not guaranteed. A poorly chosen trigger may prevent instantiation, leaving the SMT solver without crucial information. Conversely, an overly liberal trigger may cause excessive instantiations, degrading performance.

$$g(y) = y \wedge h(g(y)) = z \wedge (\forall x.\, h(g(x)) < g(h(x))) \tag{2.1}$$

To better understand this, let us explore whether $z < g(z)$ can be deduced from example 2.1 using different trigger strategies. As a first step, we split the formula into its two parts: the ground term and the quantified term.

$$\text{ground term: } g(y) = y \wedge h(g(y)) = z \tag{2.2}$$
$$\text{quantified term: } \forall x. h(g(x)) < g(h(x)) \tag{2.3}$$

trigger strat.	trigger pattern	instantiations	learned eq.
liberal	$\{g(x)\}$	$h(g(y)) < g(h(y)) \wedge$ $h(g(h(y))) < g(h(h(y))) \wedge$ $\dots$	$z < g(z)$
strict	$\{g(h(x))\}$	-	-
optimal	$\{h(g(x))\}$	$h(g(y)) < g(h(y))$	$z < g(z)$

Table 2.1: Effects of different trigger patterns on instantiations.

Table 2.1 shows the result of three different triggers and their effect on instantiations. It can be observed that a liberal trigger results in a large number of instantiations. This occurs because instantiating $g(x)$ with $x = y$ produces the term $h(g(y)) < g(h(y))$, which contains the term $g(h(y))$ that again syntactically matches the trigger $g(x)$ with $x = h(y)$. This, in turn, triggers another instantiation. Since each instantiation produces a new ground term that is subsequently instantiated again, an infinite loop, known as matching loop, occurs. In practice a threshold will prevent further instantiations to avoid endless computation.

In contrast, when the strict trigger is used, no instantiations are generated. This is due to the fact that the trigger does not appear in the ground term. As a result, no useful information is gained from the formula.

Finally, the optimal trigger strategy leads to a minimal number of instantiations. Only a single instantiation is required which provides the SMT solver with the necessary knowledge to deduce $z < g(z)$. This allows the solver to efficiently prove the desired property.

2.2 Viper

As already mentioned, Viper is a verification infrastructure that provides an intermediate language that supports an expressive permission model. As this model is based on implicit dynamic frames [19], the language is built around the accessibility predicate. It has the form `acc(e.f, p)`, which specifies permission to a specific heap location, with e denoting the reference and p indicating the amount of permission between zero and one.

To specify the transfer of permissions, Viper introduces `inhale` and `exhale` statements. These primitives, analogous to `assume` and `assert`, manage the gain and loss of permissions. An `inhale` operation grants permissions and assumes the logical constraints, while exhale removes permissions and verifies constraints. The example in Listing 2.1 showcases the usage of those primitives.

```
1 field val: Int
2 var x: Ref
3 inhale acc(x.f, write) && x.f == 0
4 exhale acc(x.f, 1 / 2) && x.f == 0
```

Listing 2.1: Demonstration of Viper's permission aware statements. The `inhale` statement gains full permission to the location of the field `x.f` and assumes that its value is equal to zero. Conversely, the `exhale` statement removes half of the permission and verifies that the value is indeed zero.

2.2.1 Predicates

A key feature of Viper for reasoning about unbounded data structures is its support for predicates. In separation logic, predicates serve as abstractions over resources and constraints. Additionally, they are treated as resources themselves, allowing them to be inhaled or exhaled. Viper supports recursively defined predicates, as shown in Listing 2.2.

To control recursion, Viper treats predicates as iso-recursive. Unlike equi-recursive predicates, where an instance is equivalent to its body, iso-recursion distinguishes between a predicate instance and its body. To manage this distinction, Viper introduces two primitives: `fold` and `unfold`. The `fold` statement replaces a predicate body with its corresponding predicate instance, abstracting away details of the underlying structure. Conversely, `unfold` expands a predicate instance into its body.

```
field val: Int
field next: Ref
predicate linked_list(l : Ref) {
    acc(l.val) && acc(l.next) &&
    (l.next != null ==> linked_list(l.next))
}
```

Listing 2.2: Typical abstraction of a linked list in Viper. The predicate `linked_list` defines access permissions for the `val` and `next` fields of each node.

The following example illustrates how the predicate defined in Listing 2.2 can be unfolded.

```
method foo() {
    var l: Ref
    inhale linked_list(l)
    assert acc(l.val) // fails
    unfold linked_list(l)
    assert acc(l.val) // succeeds
}
```

Listing 2.3: In Viper, an explicit `unfold` statement is required to access the predicate's fields.

Additionally, predicates can be folded and unfolded with fractional permission amounts, such as `fold acc(linked_list(l), 1 / 2)`. In such cases, the permission amounts of access predicates within the predicate's body are scaled accordingly. Further details on this can be found in [8].

2.2.2 Iterated separating conjunctions

While predicates are well-suited for abstracting data structures that are traversed in a structured manner, they lack the flexibility needed for random-access structures like arrays. In such cases iterated separating conjunctions (ISCs), a concept that allows expressing permission to an unbounded set of heap locations, are better suited.

Viper supports this concept, shown in Listing 2.4, by allowing access predicates, such as `acc(e.f)`, to occur within quantified expressions. This way the receiver `e` can be defined in terms of the quantified variables. The only restriction is that `e` must be injective with respect to the quantified variable. This constraint is necessary for automated reasoning and is explained later in this chapter, as well as in [15].

```
field val: Int
function e(x: Int) Ref
method foo() {
    var rs: Seq[Ref]
    var x: Int
    inhale e(x) in rs
    inhale forall i: Int ::
        e(i) in rs ==> acc(e(i).val, write)
    assert acc(e(x).val, write) // succeeds
}
```

Listing 2.4: A simple example to demonstrate the usage of an ISC in Viper. Note that the function e is assumed to be injective.

2.2.3 Silicon

Silicon is one of Viper's two backends and is based on symbolic execution as described in Section 2.1.4. To verify a Viper program, it tracks its state through a path condition stack, which manages all constraints encountered up to a specific point. Additionally, a symbolic heap stores all accessible heap locations and their associated values in form of heap chunks.

ISC algorithm

To support ISCs in Viper, Silicon uses a generalized form of the heap chunks discussed in Section 2.1.4. The following explanation describes the algorithm behind the canonical form of ISCs. A more detailed description of the general form, i.e. multiple quantified variables, along with its extension to predicates and magic wands, can be found in Schwerhoff's PhD thesis [18].

The canonical form of an ISC is defined as follows:

$$\forall i \in \mathcal{T}.\, c(i) \implies acc(e(i).f, p(i)) \tag{2.4}$$

where:

- $c(i) : \mathcal{T} \to Bool$ is a condition function that determines whether a certain condition holds for a given index $i \in \mathcal{T}$,
- $e(i) : \mathcal{T} \rightharpoonup Ref$ is a partial function mapping each index $i \in \mathcal{T}$ for which $c(i)$ holds to a receiver expression (a reference),
- $p(i) : \mathcal{T} \rightharpoonup Perm$ is a partial permission function that assigns a permission amount to each index i for which $c(i)$ holds.

The goal is storing permissions provided by the ISC into a heap chunk of the form $[r.f \to v_2 \,\#\, p_2]$. Since multiple values need to be stored rather than a

single one, the value function v_2 must be lifted to a mapping from references to values, i.e. $v_2 : Ref \to \mathcal{T}$. Similarly, the permission function must be lifted to $p : Ref \to Perm$. Additionally, the permission amount depends on the condition c as well, resulting in the final term of the form $c_2(r)\ ?\ p_2(r) : 0$.

Given these new definitions, the quantified heap chunk representation becomes:

$$[r.f \to v_2(r)\ \#\ c_2(r)\ ?\ p_2(r) : 0] \tag{2.5}$$

So far, this generalization can only be used if the quantified variable is of type Ref and $e(i) = i$, meaning e is the identity function. Since this is usually not the case, a very clever trick is used. Under the assumption that the receiver function e is injective with respect to the quantified variable i, a bijective function can be obtained by restricting the codomain, e.g. $e : \mathcal{T} \to e(\mathcal{T})$ rather than $e : \mathcal{T} \to Ref$. Let e^{-1} denote the inverse function of this restricted bijective function. The quantified heap chunk definition can be updated to:

$$[r.f \to v(r)\ \#\ img(r) \wedge c(e^{-1}(r))\ ?\ p(e^{-1}(r)) : 0] \tag{2.6}$$

To ensure correctness, the inverse function e^{-1} is defined through axioms that are added to the path conditions. Consequently, the chunk representation is also constrained by the image function $img(r)$ to maintain consistency.

Since e^{-1} is only defined on the image of e, its axioms define a partial function for those values. The required axioms are given by:

$$\forall x : \mathcal{T}.\ c(x) \implies img(e(x)) \wedge e^{-1}(e(x)) = x \tag{2.7}$$

$$\forall r : Ref.\ c(e^{-1}(r)) \wedge img(r) \implies e(e^{-1}(r)) = r \tag{2.8}$$

Equations 2.7 and 2.8 define the constraints necessary to ensure that e^{-1} behaves as the inverse of e on its codomain and is underspecified on the rest of the values.

Next, Silicon's algorithms that operate on these quantified heap chunks will be described. These include inhaling, exhaling, and reading. The following algorithms are presented in a simplified form, assuming that all chunks describe the same field. Even though the actual implementation accounts for cases where multiple fields are involved, this aspect is not essential for this thesis.

Algorithm 1 inhale($h_0, \pi_0, \forall x : \mathcal{T}.\ c(x) \implies acc(e(x).f, p(x))$)

1: ▷ injectivity check
2: **assert** $\pi_0 \models \forall y_1, y_2 : \mathcal{T}.\ c(y_1) \wedge c(y_2) \wedge p(y_1) > 0 \wedge p(y_2) > 0 \wedge e(y_1) = e(y_2) \implies y_1 = y_2$
3: **var** $e^{-1} : e(Ref) \to \mathcal{T}$ ▷ inverse function of e
4: **var** $img : Ref \to Bool$ ▷ image function of e
5: ▷ axiomatizing functions
6: **var** $ax_1 := \forall r : Ref.\ c(e^{-1}(r)) \wedge img(r) \implies e(e^{-1}(r)) = r$
7: **var** $ax_2 := \forall x : \mathcal{T}.\ c(x) \implies img(e(x)) \wedge e^{-1}(e(x)) = x$
8:
9: **var** $v : Ref \to \mathcal{T}$ ▷ define fresh value map
10: ▷ define fresh heap chunk
11: **var** $ch := [r.f \to v(r)\ \#\ img(r) \wedge c(e^{-1}(r))\ ?\ p(e^{-1}(r)) : 0]$
12:
13: **var** $h_1 := h_0 \cup_m \{ch\}$ ▷ update heap
14: **var** $\pi_1 := \pi_0 \cup \{ax_1, ax_2\}$ ▷ update path conditions
15:
16: **return** (h_1, π_1) ▷ return the updated heap and path conditions

A simplified version of Silicon's inhale algorithm is presented in Algorithm 1. It declares two new functions to represent the inverse and its image function of e. These functions are then axiomatized as described earlier.

Next, a fresh value map is defined and the quantified heap chunk is constructed. Finally, the new chunk is added to the heap, and the path conditions are extended with the axioms. Note that the heap is a multiset. To represent heap updates, we use the multiset union notation $\cup_m$.

Exhaling permissions, as shown in Algorithm 2, closely resembles the process of inhaling permissions, with only one key difference. Instead of adding a new chunk, permissions are removed by calling `remove()`, which returns an updated heap.

Updating the heap to reflect removed permission, as shown in Algorithm 3, involves iterating over all chunks and reducing their permission amounts accordingly. In order to do this correctly two key points need to be considered. First, it must be guaranteed that no chunk has more permissions removed than it originally held. This is why p_{curr} is capped by the `min` function. Second, it must be tracked how much permission has already been removed in previous iterations, which is why p_{need} is defined recursively.

As a final step, each iteration checks whether the exact permissions have already been removed. If this condition is met, the modified and unchanged parts of the heap are combined, and the updated heap is returned. If not,

Algorithm 2 exhale($h_0, \pi_0, \forall x : \mathcal{T}.\ c(x) \implies acc(e(x).f, p(x))$)

1: ▷ injectivity check
2: **assert** $\pi_0 \models \forall y_1, y_2 : \mathcal{T}.\ c(y_1) \wedge c(y_2) \wedge p(y_1) > 0 \wedge p(y_2) > 0 \wedge$
$e(y_1) = e(y_2) \implies y_1 = y_2$
3:
4: **var** $e^{-1} : e(Ref) \to T$ ▷ inverse function of e
5: **var** $img : Ref \to Bool$ ▷ image function of e
6: ▷ axiomatizing functions
7: **var** $ax_1 := \forall r : Ref.\ c(e^{-1}(r)) \wedge img(r) \implies e(e^{-1}(r)) = r$
8: **var** $ax_2 := \forall x : \mathcal{T}.\ c(x) \implies img(e(x)) \wedge e^{-1}(e(x)) = x$
9:
10: **var** $v : Ref \to \mathcal{T}$ ▷ define fresh value map
11: ▷ define fresh heap chunk
12: **var** $ch := [r.f \to v(r)\ \#\ img(r) \wedge c(e^{-1}(r))\ ?\ p(e^{-1}(r)) : 0]$
13:
14: **var** $\pi_1 := \pi_0 \cup \{ax_1, ax_2\}$ ▷ update path conditions
15: **var** $h_1 := \textbf{remove}(h_0, \pi_1, (\lambda r.\ ch.p(r)))$ ▷ remove permissions
16:
17: **return** (h_1, π_1) ▷ return the updated heap and path conditions

Algorithm 3 remove(h_0, π_0, p)

1: **var** $h_1 := \{\{\}\}$ ▷ h_1 is an empty multiset
2: **var** $h_2 := h_0$ ▷ for early stopping
3: **var** $p_{need} := p$ ▷ permission to remove $p : Ref \to Perm$
4:
5: **for** $[r.f \to v_i(r)\ \#\ p_i(r)] \in_m h_0$ **do** ▷ iterate through all chunks
6:
7: **var** $p_{curr} := \lambda r.\ \min(p_i(r),\ p_{need}(r))$
8: **var** $p_{need} := \lambda r.\ p_{need}(r) - p_{curr}(r)$ ▷ update consumed permissions
9:
10: $h_1 := h_1\ \cup_m \{\{[r.f \to v_i(r)\ \#\ p_i(r) - p_{curr}(r)]\}\}$ ▷ add updated chunk
11: $h_2 := h_2 \setminus_m \{\{[r.f \to v_i(r)\ \#\ p_i(r)]\}\}$ ▷ unchanged part of heap
12:
13: **if** $\pi_0 \models \forall r.p_{need}(r) = 0$ **then** ▷ early stopping
14: **return** $(h_1 \cup_m h_2)$
15: **end if**
16: **end for**
17:
18: **assert** $\pi_0 \models \forall r.p_{need}(r) = 0$ ▷ sufficient permissions consumed check
19: **return** (h_1) ▷ return the updated heap

by the end of the last iteration, it must be successfully proven that the exact amount of permissions has been removed.

An important fact about this algorithm is that each iteration increases the complexity of p_{need}. This means that stopping later can significantly impact the overall complexity of the resulting permission expressions. Hence, the algorithm's performance depends heavily on the order in which chunks are processed.

To read values from quantified chunks, two steps must be performed. First, it must be verified that a positive amount of permission is available. Second, all values maps are combined into a single map. Both steps are shown in Algorithm 4.

Algorithm 4 summarize(h_0)

1: **var** $v : Ref \to \mathcal{T}$ ▷ define fresh value map
2: **var** $def := \emptyset$ ▷ definitional axioms
3: **var** $p := \lambda r.\ 0$ ▷ combined permission amounts
4:
5: **for** $[r.f \to v_i(r)\ \#\ p_i(r)] \in h_0$ **do** ▷ iterate through all chunks
6: ▷ combine values
7: **var** $def := def\ \cup\ \{\forall r.\ 0 < p_i(r)\ \implies\ v(r) = v_i(r))\}$
8: **var** $p := \lambda r.\ p(r) + p_i(r)$ ▷ combine permissions
9: **end for**
10: **return** (v, def, p) ▷ return combined value map and its definitional axioms

To check sufficient permissions, the algorithm iterates over all chunks and defines the sum of all permissions provided by each chunks. Checking permissions only involves proving that $p(r) > 0$, where r denotes the location being accessed. In order to summarize the value maps, a fresh value map is defined and a constraint is added to enforce equality between the new map and each chunk's map.

Note that a single summarization adds an additional quantified constraint for each chunk to the path conditions. If invoked multiple times, it results in more complex path conditions.

2.2.4 Carbon

Carbon, Viper's other backend, employs the VCG technique described in section 2.1.3, including two maps for the program's heap and the state of resource permissions. Specifically, the two mathematical maps are defined

as follows: a $Heap : (Ref, field) \rightarrow value$, which is used to reason about values and a $Mask : (Ref, field) \rightarrow Perm$, which is used to represent permissions.

Unlike Silicon, which directly translates Viper programs into logical constraints, Carbon converts them into the external Boogie verification language [12]. The Boogie tool then generates the necessary logical expressions which are evaluated by the SMT solver to determine if a program adheres to its specified properties. Listing 2.5 showcases an example of a simplified version of this translation.

```
field val: Int
var r: Ref

inhale acc(r.val)
r.val := 1
exhale acc(r.val)
```

```
type Perm = real;
type Ref;
type Field T;
type HeapType =
    <T> [Ref, Field T] T;
type MaskType =
    <T> [Ref, Field T] Perm;
var Mask: MaskType;
var Heap: HeapType;

axiom (forall <T> Mask: MaskType,
    o: Ref, f: (Field T) ::
    { Mask[o, f] }
    (Mask[o, f] >= 0 && Mask[o, f] <= 1)
);

Mask := Mask[r, val:= 0]; //initial mask
const val: Field int
var r: Ref;
Mask := Mask[r, val:= Mask[r, val] + 1];
Heap := Heap[r, val:= 1];
assert Mask[r, val] >= 1;
Mask := Mask[r, val:= Mask[r, val] - 1];
```

Listing 2.5: The left side shows a simple Viper example, while the right side presents a simplified translation into Boogie. Maps are used to represent the heap and the permission mask. Permissions are constrained to be between zero and one, as specified by the axioms in the Boogie translation. In Viper, inhaling permissions requires updating the mask, and assigning values requires updating the heap. When permissions are exhaled, a check is performed to ensure that enough permissions are available.

Chapter 3

At-most-n predicates

In this chapter, a new feature for Viper's iso-recursive predicates is introduced. This includes a brief motivation, a general overview of the concept, and its implementation in both of Viper's backends.

3.1 Motivation

In order to prove or disprove aliasing, Viper relies on the separating conjunction, meaning that `acc(r1.f, write)` $*$ `acc(r2.f, write)` immediately implies that `r1` $\neq$ `r2`. However, when these access predicates are abstracted by iso-recursive predicates, then `acc(foo(r1), write)` $*$ `acc(foo(r2), write)` does not imply `r1` $\neq$ `r2` anymore. The reason is that Viper does not enforce an upper bound on the maximum permission that can be held for a predicate instance.

```
1 field f : Int
2 predicate foo(r : Ref) {
3     acc(r.f)
4 }
5 method main(r1 : Ref, r2 : Ref) {
6     inhale foo(r1)
7     inhale foo(r2)
8     assert r1 != r2 // fails
9 }
```

Listing 3.1: An example in Viper where disjointness cannot be proven because access predicates are hidden inside iso-recursive predicates.

Aliasing in Listing 3.1 cannot be disproven because the access predicates `acc(r1.f, write)` and `acc(r2.f, write)` are abstracted by the predicates `foo(r1)` and `foo(r2)`. Even though we can resolve the issue by unfolding

both predicates, it introduces two problems. First, it requires two additional lines of code, which is inconvenient. Second, it increases the verifier's workload, potentially causing significant overhead, especially when dealing with predicates that have a large footprint and complicate heap reasoning.

This highlights the need for a mechanism to annotate predicates, such that upper bounds on predicate's permissions can be enforced. Such an annotation would eliminate the need to unfold predicates to prove disjointness, improving both efficiency and usability.

3.2 Approach

To address the previously mentioned problem, we introduce an upper bound on the predicate's permission amount that can be held. This feature, referred to as at-most-n predicates, consists of three parts: an annotation to specify the upper bound, a validity check to ensure its correctness, and logical constraints to encode the upper bound. This concept builds upon a proposal from a previous bachelor's thesis [21], which explored performance improvements related to this issue.

```
1 field f : Int
2 predicate foo(r : Ref) limit write {
3     acc(r.f)
4 }
5 method main(r1 : Ref, r2 : Ref) {
6     inhale foo(r1)
7     inhale foo(r2)
8     assert r1 != r2 // succeeds
9 }
```

Listing 3.2: Example of the new at-most-n predicate feature in Viper. The predicate definition is followed by the keyword `limit` and a permission literal, e.g. `write`.

Given an upper bound, we can assume that the bound holds at the beginning of every method, meaning we add an additional constraint to explicitly describe the upper bound.

The purpose of Listing 3.3 is to provide an intuition for this concept. The actual implementation varies significantly depending on the backend. Further details can be found in Section 3.3.

```
1 field f : Int
2 predicate foo(r : Ref) limit write {
3     acc(r.f)
4 }
5 method main(r1 : Ref, r2 : Ref) {
6     // assumption of upper bound
7     inhale forall r: Ref :: perm(foo(r)) <= write
8     inhale foo(r1)
9     inhale foo(r2)
10    assert r1 != r2 // succeeds
11 }
```

Listing 3.3: Illustration of the conceptual assumption of an upper bound in Viper. Every method has an additional constraint, such as the one on line 7, which allows proving `r1 != r2`.

3.2.1 Annotation

We extended the Viper language with a new keyword `limit`, allowing users to specify a maximum amount of a predicate's permission that can be held. The `limit` keyword appears after the predicate's signature and is followed by a permission literal that defines the bound.

```
1 field f : Int
2 predicate foo(r : Ref) limit write {
3     acc(r.f)
4 }
```

Listing 3.4: Users can define an upper bound on predicates using the `limit` keyword, followed by a permission value.

This feature provides a structured way to control upper bounds, as shown in Listing 3.4.

Alternatively, the upper bound could allow heap-dependent values for which the predicate's body provides permission. However, the use cases for such an approach are not clear, and it significantly increases the complexity of the overall concept.

Listing 3.5 illustrates a heap-dependent upper bound. In this case, the bound `1 / x.p` is defined in terms of the field `x.p`, for which `foo`'s body has permission.

We considered allowing such heap-dependent bounds. However, two significant challenges arise. First, the practical usefulness of this feature remains unclear, as real-world use cases requiring heap-dependent bounds are rare. Second, implementing such a feature introduces complexity and potential unsoundness as self-referential constraints on permission bounds can lead to

```
field f : Int
field p : Perm
predicate foo(x: Ref) limit 1 / x.p {
    acc(x.p, wildcard) &&
    none <= x.p &&
    acc(x.f, x.p)
}
```

Listing 3.5: Example of a heap-dependent upper bound in Viper. This is currently not allowed due to potential unsoundness.

inconsistencies in verification. For these reasons, we decided against allowing heap-dependent upper bounds.

3.2.2 Validity check

In order to soundly introduce at-most-n predicates, Viper must ensure that the user provided upper bound is valid. Clearly, this is only possible if the predicate is not abstract. Upper bounds for abstract predicates are simply assumed to be correct. In such cases it is the users responsibility to not introduce any inconsistencies. In all other cases predicate's upper bounds must be validated.

```
field f : Int
predicate foo(r : Ref) limit 1/2 {
    acc(r.f)
}
method main(r : Ref) {
    inhale acc(r.f)
    fold foo(r)
    // contradicting conditions after the fold:
    // perm(foo(r)) == write and perm(foo(r)) <= 1/2
    assert false // succeeds
}
```

Listing 3.6: The predicate `foo` is annotated with an incorrect upper bound of $1/2$. When the predicate is folded with a permission exceeding this bound, a contradiction is introduced. As a result the verifier successfully proves `false`.

If the correctness of the upper bound is guaranteed, the issue demonstrated in Listing 3.6 can no longer occur. Any attempt to fold a predicate with a permission greater than its upper bound will always fail as the required permissions for its body cannot exist.

The upper bound property is global and needs to be proven only once per predicate definition, similar to its self-framing proof. This property is

proven by contradiction. If a client were able to hold permissions exceeding the predicate's declared upper bound, unfolding would always result in a contradiction.

```
field f : Int
predicate foo(r1 : Ref) limit write {
    acc(r1.f)
}
method proof(r1 : Ref) {
    var up: Perm
    inhale ub > write
    inhale acc(foo(r1), ub)
    unfold acc(foo(r1) ub)
    assert false
}
```

Listing 3.7: Demonstration of the upper bound validity check in Viper. The method assumes a permission amount greater than the upper bound for predicate `foo`. If the subsequent unfolding of `foo` introduces a contradiction, meaning `assert false` succeeds, the upper bound is valid.

3.2.3 Multiple arguments

So far, all predicates in previous examples had only one argument. The reason for this is that the concept of at-most-n predicates gets significantly more complex with more arguments.

```
field f : Int
predicate foo(r1: Ref,
    r2: Ref) limit write {
    acc(r1.f) &&
    acc(r2.f, 1/2)
}

method main(r1: Ref, r2: Ref,
    r3: Ref, r4: Ref) {
    inhale foo(r1, r3)
    inhale foo(r2, r4)
    assert r1 != r2 // fails
}
```

```
field f : Int
predicate foo(r1: Ref,
    i: Int) limit write {
    acc(r1.f) &&
    r1.f > i
}

method main(r1: Ref,
    r2: Ref) {
    inhale foo(r1, 0)
    inhale foo(r2, 1)
    assert r1 != r2 // fails
}
```

Listing 3.8: Examples of predicates with multiple arguments and an upper bound. In both cases, the disjointness check fails even though it is theoretically correct.

In order to understand why the upper bound does not help to prove disjoint-

ness in Listing 3.8, we first need to define predicate equality. Two predicate instances are equal if and only if all their arguments are equal.

With this in mind, in the example on the right hand side, `foo(r1, 0)` and `foo(r2, 1)` are not the same predicate instance, not even under the assumption of `r1 = r2`. Hence, the upper bound does not impose any restrictions.

The same problem happens on the left hand side. The core problem is that the upper bound depends on both arguments. This dependency prevents the verifier to conclude disjointness.

In theory, this problem can be resolved by allowing the upper bound to depend only on a subset of the predicate's arguments. Additionally, users would need the ability to specify multiple upper bounds. The updated annotation would be as follows:

```
1 field f : Int
2 predicate foo(r1 : Ref, r2: Ref)
3     limit {r1} write
4     limit {r2} 2
5 {
6     acc(r1.f) && acc(r2.f, 1 / 2)
7 }
```

Listing 3.9: A user-annotated predicate with two different upper bounds, each depending on a different subset of arguments.

While the annotation shown in Listing 3.9 would enable proving the failing assertions in Listing 3.8, at the time of writing it is not clear, how such an annotation can be implemented.

One possible approach is to introduce an additional abstract predicate for each upper bound, using the same set of arguments specified in the annotation. These abstract predicates would then mirror all operations performed on the original predicate as illustrated in Listing 3.10.

While this approach works for single instances, it presents a flaw when applied to ISCs. As explained in Section 2.2.2, ISCs must be injective with respect to their quantified variables. However, restricting a predicate to fewer arguments while mirroring the same operations results in injectivity issues.

Listing 3.11 illustrates the injectivity problem with ISCs. Since `foo1` and `foo2` depend only on `r1` and `r2`, respectively, they are not injective with respect to both `r1` and `r2`. As a result, an unexpected verification error would occur to the user.

This highlights that implementing an approach that supports multiple arguments, as previously described, is significantly more complex than initially

```
predicate foo1(r1: Ref) limit write
predicate foo2(r2: Ref) limit 2

method main(r1: Ref, r2: Ref, r3: Ref, r4: Ref) {
     // assumption of upper bounds
    inhale forall r: Ref :: perm(foo1(r)) <= write
    inhale forall r: Ref :: perm(foo2(r)) <= 2
    inhale foo(r1, r3)
    inhale foo1(r1) && foo2(r3) // mirror inhale operation
    inhale foo(r2, r4)
    inhale foo1(r2) && foo2(r3) // mirror inhale operation
    assert r1 != r2 // succeeds
}
```

Listing 3.10: New abstract predicates are added for each upper bound defined in Listing 3.9. The upper bound can then be encoded using these predicates. This allows to prove the desired assertion.

```
function c(r1: Ref, r2: Ref): Bool
method main() {
    inhale forall r1: Ref, r2: Ref ::
        c(r1,r2) ==> acc(foo(r1, r2))
    // mirror ISC with abstract predicates

    inhale forall r1: Ref, r2: Ref ::
        c(r1,r2) ==> acc(foo1(r1))
    inhale forall r1: Ref, r2: Ref ::
        c(r1,r2) ==> acc(foo2(r2))
    // error: ISCs are not injective
}
```

Listing 3.11: Injectivity issue with ISCs when using abstract predicates to mirror the original ISC.

anticipated. Therefore, we opted for the simpler solution that allows only a single upper bound per predicate which always depends on all its arguments.

3.3 Implementation

This section details the implementation of the previously discussed approach and is divided into three parts. The first part covers the modifications to the verification language, the second part describes the implementation in Carbon, followed by the final part about the implementation in Silicon.

All examples in this section are encoding the following predicate:

```
predicate foo(r: Ref) limit write {
    acc(r.f, write)
}
```

3.3.1 Verification language

In order to support at-most-n predicates, two components must be added to Viper's verification language. First, the abstract syntax tree (AST) must be extended to include the new `limit` keyword. Second, a new error message must be added to report verification failures when the validity check does not succeed.

```
limit ::= "limit" exp

predicate ::=
    "predicate" ident formal-args [limit] ["{" exp "}"]
```

Listing 3.12: Extension to Viper's AST.

Listing 3.12 shows the modifications made to the Viper AST. These changes include the new keyword `limit` as well as an updated predicate signature. Note that the upper bound is only optional and can be omitted. The restriction that the expression following `limit` must be of type `Perm` is then enforced during parsing.

```
field f : Int
predicate foo(r1: Ref) limit write {
    acc(r1.f, 1/2)
}
// Output: Upper bound might not hold. Upper bound write for
//         predicate foo might not hold.
```

Listing 3.13: Example of the new verification error message for an invalid upper bound in Viper. The comment displays the error message that appears when an invalid upper bound is provided.

Listing 3.13 demonstrates the new verification error for invalid upper bounds in Viper. In this example, the upper bound `write` is not valid for the predicate `foo`, causing the validity check to fail and triggering a verification error message which is shown in the comment.

This includes all changes made to Viper's verification language.

3.3.2 Carbon

Carbon, as explained earlier, translates Viper to Boogie. Hence, we need to encode the validity check and the upper bound into Boogie to support at-most-n predicates.

Validity check

The encoding of the validity check is straightforward translation from Viper to Boogie. The following example demonstrates this translation in more detail.

```
1  procedure fooUpperBound(r: Ref) returns ()
2    modifies Mask;
3  {
4    var limit: Perm where limit > write;
5
6    // -- Check upper bound of predicate foo
7      // -- Initializing the state
8        Mask := ZeroMask;
9
10     havoc limit;
11     assert {:msg "Upper bound might not hold.
12         Fraction limit might be negative."}
13       limit >= NoPerm;
14     assume limit > NoPerm ==> x != null;
15     Mask := Mask[r, bar:=Mask[r, bar] + limit];
16     assert {:msg "Upper bound might not hold.
17         Upper bound write for predicate foo might not hold."}
18       false;
19 }
```

Listing 3.14: Translation of the upper bound validity check from Viper to a Boogie procedure for the predicate `foo`.

Listing 3.14 presents a simplified version of the upper bound validity check, closely resembling the actual implementation. It includes the generation of a new procedure in Boogie which is allowed to modify the global permission mask.

Within the procedure, we define a new variable `limit` of type `Perm` which is constrained to be strictly greater than the upper bound. The state is then reset, which includes setting the entire permission mask to zero. Following this, the permission variable is assigned an arbitrary value (havocked). Next, we need to ensure that this assigned value is nonnegative.

Finally, we update the permission mask according to the access predicates within predicate's body. In this case it contains only a single field named `f`. This step introduces a contradiction given a valid upper bound, ensuring that the last assertion succeeds.

Encoding

Once again, the translation from Viper to Boogie in Carbon closely follows the intuitive explanation given in Section 3.2. The only minor difference is that the encoding is enforced through a single global constraint, rather than being applied individually to each method.

```
1 axiom (forall Mask: MaskType, r: Ref ::
2   { Mask[null, foo(r)] }
3   Mask[null, foo(r)] <= write
4 );
```

Listing 3.15: Global constraint to encode `foo`'s upper bound in Boogie.

Listing 3.15 presents the global axiom used to encode an upper bound by restricting the permission mask.

This concludes the implementation of the at-most-n predicates feature in Carbon.

3.3.3 Silicon

Silicon translates Viper code directly into SMT-LIB [4]. As stated in Section 2.2.3, constraints are stored in path conditions for each control flow branch. Consequently, a global axiom, as used in Carbon, is not implementable, requiring a different way of encoding the upper bound.

Validity check

Even though, the translated language is different, the translation of the validity check is very similar in Silicon to Carbon.

Although SMT-LIB is different to Boogie, the translation of the validity check remains conceptually similar. Listing 3.16 illustrates a simplified SMT-LIB script produced by Silicon.

Silicon's path conditions are encoded in SMT-LIB via the `assert` keyword. In the example shown in Listing 3.16, three path conditions are assumed. First, the permission held of the predicate instance exceeds its upper bound, $(> \mathit{limit}\ 1)$. Second, the permission of field `f` is non-negative, $(<= 0\ \mathit{limit})$. Third, the permission of field `f` is at most one, $(<= \mathit{limit}\ 1)$. Clearly, the first

```
1  (declare-sort $Ref 0)
2  (define-sort $Perm () Real)
3
4  (declare-const xs $Ref)
5  (declare-const limit $Perm)
6
7  (assert (> limit 1)) ; attempt to hold permission
       greater than upper bound
8  (assert (<= 0 limit)) ; permisssion for field f >= 0
9  (assert (<= limit 1)) ; permission for field f <= 1
10 (check-sat)
11 ; unsat
```

Listing 3.16: Simplified SMT script showcasing Silicon's upper bound validity check.

and and third conditions contradict each other, proving that the upper bound is valid.

Encoding

In Silicon, the implementation of the encoding of an upper bound is very different than the conceptional explanation in 3.2. The upper bound is not globally assumed but rather added on a per chunk basis. In order to do this, we need to divide the chunks into single heap chunks and quantified heap chunks.

Starting with single heap chunks, Silicon defines a set of properties that can be applied to each chunk. These properties are functions that, given a chunk, return a constraint describing certain restrictions on that chunk. For example, one such property ensures that the permission amount represented by a chunk must be at least zero.

$$\lambda\ ch.\ ch.p >= 0 \tag{3.1}$$

The property in Equation 3.1 can be applied whenever necessary by evaluating the function for each chunk and adding the resulting constraints to the path conditions. This ensures that the SMT solver is aware of the chunk's constraints.

Building on this concept of properties, we can define an new property that describes the upper bound restriction.

$$\lambda\ ch.\ hasUpperBound(ch) \implies perm(ch) \leq ch.upperBound \tag{3.2}$$

Equation 3.2 defines such an upper bound property. Note that the upper bound is optional, meaning an implication is used which is trivially true if no upper bound is provided.

As a final step, we define the separating conjunction, ensuring that if the sum of two predicate instances exceeds the upper bound, their arguments must be distinct. This is also defined as a property. However, unlike previous properties that restrict individual chunks, this one describes relations between chunks.

$$\begin{aligned} \lambda\ ch_1, ch_2.\ & hasUpperBound(ch_1) \implies \\ & perm(ch_1) + perm(ch_2) > ch_1.upperBound \implies \\ & ch_1.args \neq ch_2.args \end{aligned} \tag{3.3}$$

Adding the property from Equation 3.3 to the path conditions requires evaluating the function for all possible combinations of chunks. This process introduces n^2 new constraints for n chunks.

With this, the implementation for single heap chunks is complete.

Next, we focus on quantified chunks and their encoding, which involves iterating over all chunks and adding a constraint for each chunk to the path conditions.

There are two possible approaches for achieving this. The first approach utilizes the predicate as trigger, allowing its definition to be used directly. The second approach relies on the inverse function of the chunk when a predicate trigger is unavailable.

$$\begin{aligned} \forall r : Ref.\ \{foo(r)\}\ & ch_i.p(r) > 0 \implies \\ & ch_i.p(r) \leq ch_i.upperBound \end{aligned} \tag{3.4}$$

$$\begin{aligned} \forall r : Ref.\ \{e^{-1}(r)\}\ & ch_i.p(e^{-1}(r)) > 0 \implies \\ & ch_i.p(e^{-1}(r)) \leq ch_i.upperBound \end{aligned} \tag{3.5}$$

Equation 3.4 is preferable due to its simplicity. However, its applicability depends on whether the user provided the trigger $\{foo(r)\}$ when inhaling the permissions for ch_i. If the trigger is not available, Equation 3.5 is used.

Iterating over all quantified chunks produces the set of constraints defined in Equation 3.6, which are then added to the path conditions.

$$\bigcup_{i=1}^{n} \left(hasUpperBound(ch_i) \implies [3.4 \mid 3.5] \right) \tag{3.6}$$

Note that quantified chunks add only n constraints to the path conditions, whereas single chunks require n^2. In theory, single chunks could be encoded similarly, reducing the number of constraints from n^2 to n. However, this approach would rely on quantified constraints, unlike the current implementation. Since quantified constraints are much harder to reason about, this would decrease performance despite the reduction of constrains by a factor of n.

This concludes the implementation of the at-most-n predicates feature in Silicon.

3.4 Evaluation

This section serves two main purposes. First, it demonstrates the implementation of the at-most-n predicate feature in practice. Second, it evaluates its performance impact in scenarios involving large number of predicates or particularly complex ones. The results validate the claims made earlier, showing that the feature enhances verification performance and reduces code complexity.

3.4.1 Experiment setup

The experiments were conducted on the Euler cluster at ETH Zurich, utilizing a single CPU and 8 GB of RAM. Each benchmark program was verified using both the Carbon and Silicon backend, with incremental increases in complexity. Each complexity level was verified 50 times per verifier, and the results were averaged over these runs to ensure statistical significance.

3.4.2 Examples

The following examples highlight different challenges in verification and demonstrate the impact of the proposed features.

Unfolding overhead

The first example, shown in Listing 3.17, examines a predicate that grows in complexity with an increasing number of fields.

Figure 3.1 illustrates that as the complexity of the predicate increases, unfolding leads to longer verification times. However, by imposing an upper bound, unfolding becomes unnecessary to get the non-aliasing information required to prove the final assertion, resulting in improved performance. In Carbon, the speedup is approximately 3 ± 0.05 for ten implications, while in Silicon, the speedup is around 25 for just seven implications.

```
field f0 : Int
field b0 : Int
field f1 : Int
field b1 : Int
predicate foo(r: Ref) limit write {
    acc(r.b0, write) && (r.b0 ==> acc(r.f0, write)) &&
    acc(r.b1, write) && (r.b1 ==> acc(r.f1, write))
}
method foo() {
    var a: Ref
    var b: Ref
    inhale acc(P(a), write)
    inhale acc(P(b), write)
    unfold P(a) // omitted with upper bound
    unfold P(b) // omitted with upper bound
    assert a != b
}
```

Listing 3.17: Illustration of a predicate with two implications. The complexity in this example is increased by introducing additional field definitions, along with their corresponding access predicates and implications within the predicate. The predicate is then unfolded within the method `foo` to prove disjointness between two references.

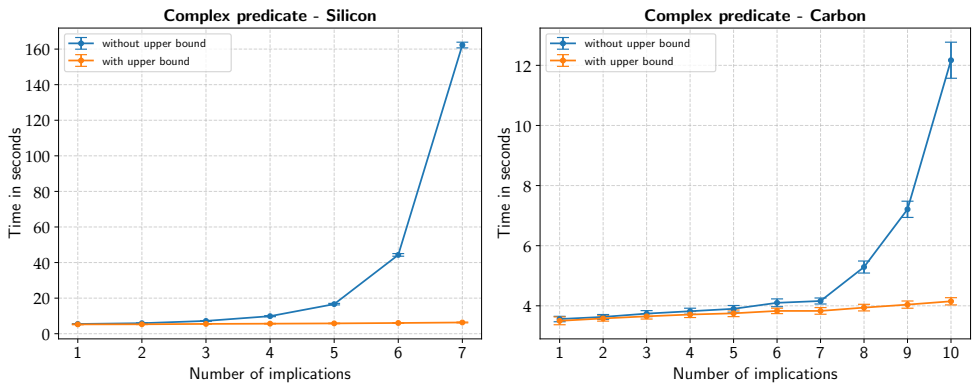

Figure 3.1: Verification times for the example in Listing 3.17 with increasing complexity. The left graph represents verification using Silicon, while the right graph represents verification using Carbon. The X-axis indicates the number of implications within the predicate, and the Y-axis represents the verification time in seconds.

In Carbon, the increased verification times arise from a harder-to-prove VC, while in Silicon, each implication within the predicate requires branching, leading to an exponential increase in the number of branches that need verification.

Branch pruning

The next example, shown in Listing 3.18, involves a method with numerous unreachable branches in the control flow.

```
field f : Int
predicate foo(r: Ref) limit write {
    acc(r.f, write)
}
method foo() {
    var a0: Ref
    var a1: Ref
    inhale acc(P(a0), write)
    inhale acc(P(a1), write)
    if (a1 == a0) {
        assert a1 == a0
    }
}
```

Listing 3.18: Illustrates a case with two references. Complexity in this example is increased by introducing additional references and separate `if`-blocks for each pair. The assertion within each `if`-block is irrelevant and trivially true.

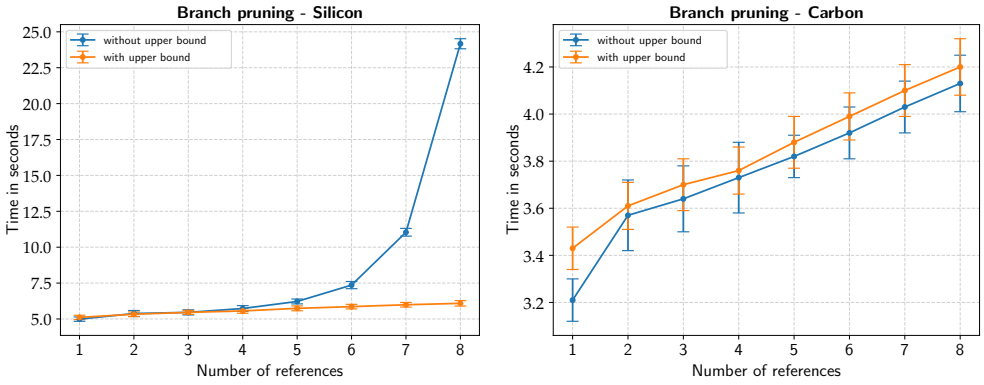

Figure 3.2: Verification times for the example in Listing 3.18 with increasing complexity. The left graph represents verification using Silicon, while the right graph represents verification using Carbon. The X-axis indicates the number of references, and the Y-axis represents the verification time in seconds.

While Carbon efficiently handles these cases even without an upper bound, resulting in a no-speedup, Silicon experiences performance degradation due to unnecessary verification of unreachable branches. As shown in Figure 3.2, the upper bound enables Silicon to recognize these branches as unreachable, improving verification times, with a speedup of approximately 4 ± 0.05 for ten references.

Multiple predicates

The final example, shown in Listing 3.19, considers multiple predicates that must be unfolded to verify disjointness.

```
field f : Int
predicate foo(r: Ref) limit write {
    acc(r.f, write)
}
method foo() {
    var a0: Ref
    var a1: Ref
    inhale acc(P(a0), write)
    inhale acc(P(a1), write)
    unfold acc(P(a0), write) // omitted with upper bound
    unfold acc(P(a1), write) // omitted with upper bound
    assert a0 != a1
}
```

Listing 3.19: Illustrates a case with two references. Complexity in this example is increased by adding additional references, `unfold` statements, and assertions that check each reference is disjoint from `a0`.

While this does not impact Silicon, Carbon struggles with an increasing number of unfolded predicates, as the permission mask becomes progressively more complex. As shown in Figure 3.3, introducing an upper bound prevents this issue because the `unfold` statements can be omitted while still proving the final assertions, resulting in improved performance with a speedup of nearly 6. Additionally, fewer lines of code are required, as explicit unfolding is no longer necessary.

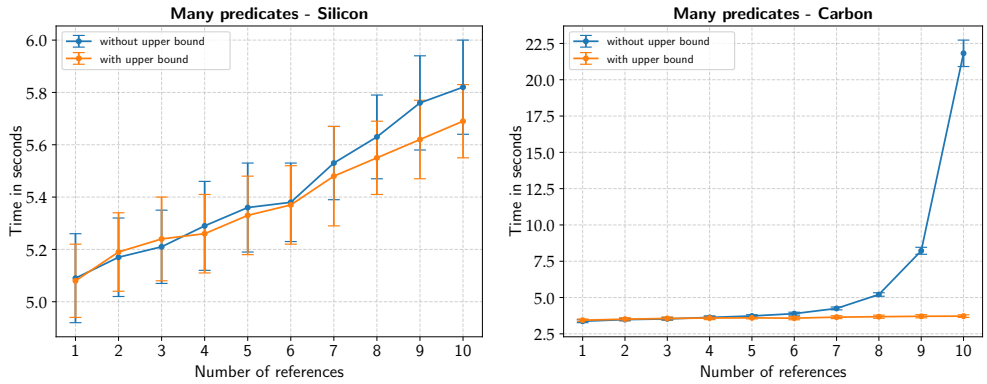

Figure 3.3: Verification times for the example in Listing 3.19 with increasing complexity. The left graph represents verification using Silicon, while the right graph represents verification using Carbon. The X-axis indicates the number of references, and the Y-axis represents the verification time in seconds.

3.4.3 Summary

The evaluation demonstrates the advantages of the proposed at-most-n predicates feature. The additional validity check does not negatively impact performance, while the upper bound assumptions can reduce verification times in certain scenarios. Overall, these findings validate the effectiveness of this feature.

Chapter 4

Greedy ISC algorithm

This chapter introduces a new greedy approach for handling ISCs. It begins by analyzing the most significant performance issues, followed by a presentation of the greedy approach and techniques to improve completeness. The chapter concludes with an evaluation of the proposed method.

4.1 Performance investigation

First, we examine potential performance issues with the current ISC algorithm to justify our focus on improving it. As briefly mentioned in Section 2.2.2, Algorithm 3 depends on the order in which the heap chunks are processed. Although Silicon already employs various optimizations to mitigate this dependency, in many programs it still results in significant overhead.

From an algorithmic perspective, with only three chunks $[ch_1, ch_2, ch_3]$, the complexity permission of ch_3 compared to ch_3 is considerable. For example, if we want to remove the $\lambda r.\ p_{init}(r)$ permission, the new permission expression $p_{new_1}, p_{new_2}, p_{new_3}$ looks the following:

$$
\begin{aligned}
p_{new_1} &:= \lambda r.\ p_1(r) - \min(p_1(r),\ p_{init}(r)) \\
p_{new_2} &:= \lambda r.\ p_2(r) - \min(p_2(r),\ p_{init}(r) - \min(p_1(r),\ p_{init}(r))) \\
p_{\text{new}_3} &:= \lambda r.\ p_3(r) - \\
&\qquad \min(p_3(r),\ p_{\text{init}}(r) - \min(p_2(r),\ p_{\text{init}}(r) - \min(p_1(r),\ p_{\text{init}}(r)))).
\end{aligned}
$$

Even with only three chunks, p_{new_3} is considerably more complex than p_{new_1}. Without any optimizations, this complexity becomes unmanageable with more than ten chunks, as demonstrated in the following example.

Adding additional arrays as arguments to `arraySum` increases the number of ISCs, increasing the likelihood of an unfavorable iteration order. Additionally,

```
1  method arraySum(a0: IArray, n: Int) returns (res:Int)
2      requires forall j: Int ::
3          0 <= j && j < n ==> acc(loc(a0, j).val, 2/3)
4      requires len(a0) == n
5      ensures forall j: Int ::
6          0 <= j && j < n ==> acc(loc(a0, j).val, 2/3)
7      ensures len(a0) == n
8  {
9      var i: Int
10     i := 0
11     res := 0
12     while(i < n )
13         invariant forall j: Int ::
14             0 <= j && j < n ==> acc(loc(a0, j).val, 2/3)
15         invariant len(a0) == n
16         invariant  0 <= i && i <= n
17     {
18         res := res + loc(a0, i).val
19         i := i + 1
20     }
21 }
```

Listing 4.1: The program computes the sum of all elements in a0. The necessary permissions are specified using ISCs.

more read accesses occur within the loop, requiring additional heap summarizations. Silicon attempts to mitigate these issues in two ways. First, it applies a heuristic to select a more suitable chunk order, and second, it uses summarization caching to reuse previously computed summaries if the heap remains unchanged. Although these heuristics can significantly improve performance, they have limitations. In some cases, they fail to sufficiently reduce overhead, leading to performance bottlenecks.

To clearly demonstrate the underlying performance problem, these heuristics are disabled via the flags `--disableChunkOrderHeuristics` and `--disableValueMapCaching`. Figure 4.1 shows that with a linear increase in ISCs, an exponential growth occurs in the number of quantifier instantiations and total verification time. Furthermore, the verification process becomes increasingly unstable, as indicated by an increase in the standard deviation.

By reducing reliance on chunk processing order and the number of axioms emitted by summarizations, Silicon could improve its ability to handle larger, more complex programs while reducing the risk of exponential runtime or instability.

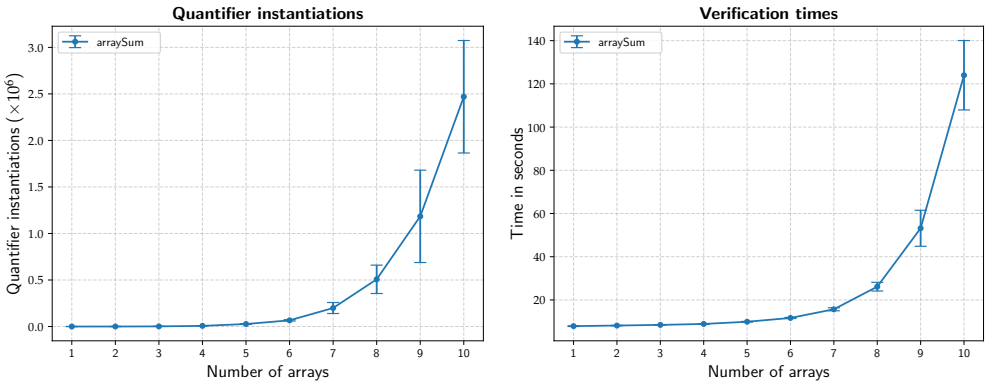

Figure 4.1: With a linear increase in ISCs, both the number of quantifier instantiations (left) and the verification time (right) rise exponentially. The standard deviation also increases, indicating more instability. The data represent the mean and standard deviation of 10 verification runs.

4.2 Greedy exhale

An optimization already implemented in Silicon splits the heap into relevant and irrelevant chunks before removing permissions. A chunk is considered irrelevant if and only if it provides permission for references that do not require permission removal. More formally, consider a chunk $[r.f \rightarrow ch.v(r) \# ch.p(r)]$ and a permission function $p : Ref \rightarrow Perm$ representing the permissions to be removed. The proof obligation for an irrelevant chunk is given by Equation 4.1.

$$\forall r : Ref.\ ch.p(r) > 0 \implies p(r) = 0 \tag{4.1}$$

Filtering out irrelevant chunks is straightforward in principle. As shown in Algorithm 5, the procedure iterates over all chunks, attempts to prove Equation 4.1, and discards each chunk if the condition holds. This avoids updating permission expressions for those discarded chunks. Note that in Silicon's implementation, Algorithm 5 is combined with Algorithm 3 to avoid iterating over the heap twice.

This optimization can be seen as an over-approximation of the set of relevant chunks. A chunk might be classified as relevant even if it is not but it will never be considered irrelevant if it is relevant. Unfortunately, in practice, proving Equation 4.1 is difficult, resulting in many irrelevant chunks still being classified as relevant.

An alternative approach would be to under-approximate the relevant chunks, meaning that all chunks are filtered out unless they contain locations from which permissions will be removed. Note that this approach is less complete

Algorithm 5 filter_relevant_chunks(h_0, p)

1: **var** $h_1 := \{\{\}\}$ ▷ h_1 is an empty multiset
2:
3: **for** $[r.f \rightarrow v_i(r) \# p_i(r)] \in_m h_0$ **do** ▷ iterate through all chunks
4:
5: **if** $\forall r : Ref.\ p_i(r) > 0 \implies p(r) = 0$ **then**
6: $h_1 := h_1 \cup_m \{\{[r.f \rightarrow v_i(r) \# p_i(r)]\}\}$ ▷ filter out irrelevant chunks
7: **end if**
8:
9: **end for**
10: **var** $h_{relevant} := h_0 \setminus_m h_1$
11: **return** $h_{relevant}$ ▷ return relevant part of heap

than an over-approximation, as relevant chunks could now be misclassified as irrelevant.

The example in Listing 4.2 highlights such an incompleteness issue, which can arise only with an under-approximating chunk-picking strategy, whereas an over-approximating strategy successfully verifies the assertion. The root cause lies in the disjunction on line 8. We cannot definitively determine which chunk grants permission for `x.f`. Hence, an over-approximation picks both chunks, enabling the assertion to be proven while an under-approximation selects neither, leading to a failed proof. With this greedy approach, there is little that can be done from an algorithmic point of view to resolve this issue.

However, it is often possible to rewrite the program in a way that explicitly indicates which chunk is needed, resulting in a successful verification, as demonstrated in Listing 4.3.

```
1  field f: Int
2  method foo() {
3      var x: Ref
4      var s1: Set[Ref], s2: Set[Ref]
5      inhale forall r: Ref ::
6          r in s1 ==> acc(r.f, write) // chunk 1
7      inhale forall r: Ref ::
8          r in s2 ==> acc(r.f, write) // chunk 2
9      inhale x in s1 || x in s2
10     assert acc(x.f, write) // permission is granted either
11                            // by chunk 1 or chunk 2
12 }
```

Listing 4.2: Example demonstrating how an under-approximating chunk-picking strategy can be less complete than an over-approximation one.

```
1  field f: Int
2  method foo() {
3      var x: Ref
4      var s1: Set[Ref], s2: Set[Ref]
5      inhale forall r: Ref ::
6          r in s1 ==> acc(r.f, write) // chunk 1
7      inhale forall r: Ref ::
8          r in s2 ==> acc(r.f, write) // chunk 2
9      inhale x in s1 || x in s2
10     if (x in s1) {
11         assert acc(x.f, write) // perm. is granted by chunk 1
12     } else {
13         assert acc(x.f, write) // perm. is granted by chunk 2
14     }
15 }
```

Listing 4.3: Rewritten Listing 4.2 to explicitly indicate the permission-granting chunk, enabling an under-approximation strategy to succeed.

Let us further explore the concept of under-approximation. To select a chunk, we need to prove Equation 4.2. Unfortunately, this equation includes an existential quantifier which is known to be challenging for SMT solvers to handle effectively in practice.

$$\exists r : Ref.\ p(r) > 0 \implies ch.p(r) > 0 \tag{4.2}$$

However, an important observation can simplify the proof obligation without significantly compromising completeness in practice. To understand this, we need to analyze the relationship between locations for which we want to remove permission and locations for which a chunk grants permission.

Let us denote $P := \{r \mid p(r) > 0\}$ and $Q := \{r \mid ch.p(r) > 0\}$ as the set of those locations. Then intersection $S := P \cap Q$ can take four different forms: it may be empty, a strict subset of both P and Q or it may be equal to either P or Q.

The last two cases are particularly interesting. In those cases, the proof obligation for selecting the chunk is significantly lower, as it no longer involves an existential quantifier. The corresponding proof obligations are presented in Equations 4.3 and 4.4, respectively.

$$\forall r : Ref.\ p(r) > 0 \implies ch.p(r) > 0 \tag{4.3}$$
$$\forall r : Ref.\ ch.p(r) > 0 \implies p(r) > 0 \tag{4.4}$$

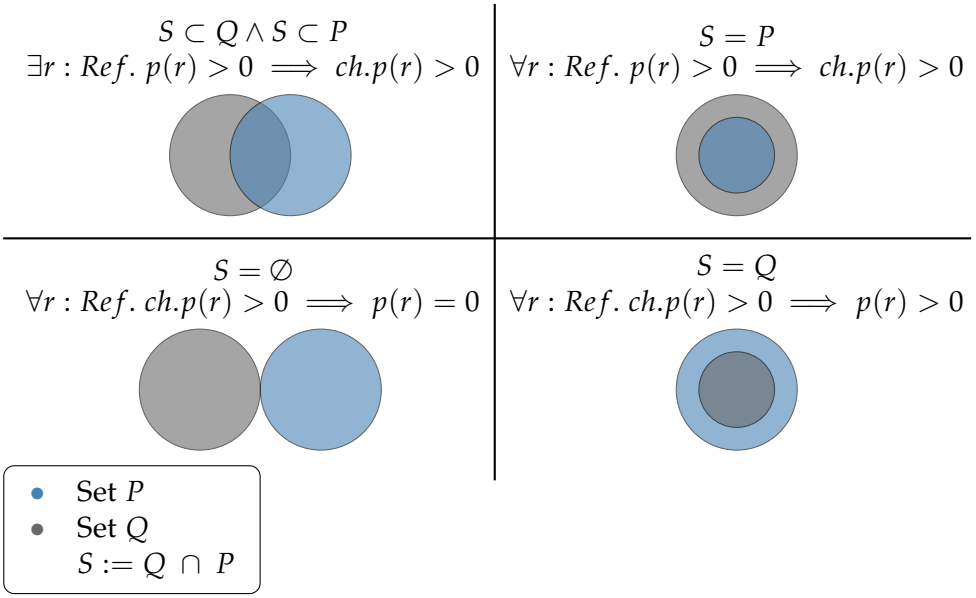

Figure 4.2: Summary of the previously discussed cases and the corresponding constraints that must be satisfied for each case to hold.

Now, the main idea of this under-approximating approach is the observation that the case in the left upper corner in Figure 4.2 is rarely encountered in practice. Usually, an ISC is used to describe permissions to locations that share a common property, such as belonging to the same array. It is very unidiomatic, for instance, to describe a subset of locations spanning two different arrays within a single ISC.

This insight leads us to the final version of the under-approximating chunk-picking algorithm. By excluding this uncommon case, we are able to develop a simple and efficient method for selecting relevant chunks. Algorithm 6 illustrates this process.

As a final step, we combine both chunk-picking strategies. First, we attempt to greedily pick chunks for permission removal. If this fails, we retry using the over-approximating chunk-picking strategy. If that fails again, we report a verification error. This approach provides the best of both worlds. A more aggressive strategy for performance gains and a fallback strategy that prioritizes completeness. Algorithm 7 combines the two strategies in a single `exhale` function.

Algorithm 6 under_approx(h_0, p)

1: **var** $h_1 := \{\{\}\}$ ▷ h_1 is an empty multiset
2:
3: **for** $[r.f \to v_i(r) \# p_i(r)] \in_m h_0$ **do** ▷ iterate through all chunks
4:
5: **if** $\forall r : Ref.\ p(r) > 0 \implies p_i(r) > 0 \lor$
$\forall r : Ref.\ p_i(r) > 0 \implies p(r) > 0$ **then**
6:
7: $h_1 := h_1 \cup_m \{\{[r.f \to v_i(r) \# p_i(r)]\}\}$ ▷ keep relevant chunks
8: **end if**
9:
10: **end for**
11: **return** h_1 ▷ return relevant part of heap

Algorithm 7 exhale($h_0, \pi_0, \forall x : \mathcal{T}.\ c(x) \implies acc(e(x).f, p(x))$)

1: ▷ injectivity check
2: **assert** $\pi_0 \models \forall y_1, y_2 : \mathcal{T}.\ c(y_1) \land c(y_2) \land p(y_1) > 0 \land p(y_2) > 0 \land$
$e(y_1) = e(y_2) \implies y_1 = y_2$
3:
4: **var** $e^{-1} : e(Ref) \to T$ ▷ inverse function of e
5: **var** $img : Ref \to Bool$ ▷ image function of e
6: ▷ axiomatizing functions
7: **var** $ax_1 := \forall r : Ref.\ c(e^{-1}(r)) \land img(r) \implies e(e^{-1}(r)) = r$
8: **var** $ax_2 := \forall x : \mathcal{T}.\ c(x) \implies img(e(x)) \land e^{-1}(e(x)) = x$
9:
10: **var** $v : Ref \to \mathcal{T}$ ▷ define fresh value map
11: ▷ define fresh heap chunk
12: **var** $ch := [r.f \to v(r) \# img(r) \land c(e^{-1}(r))\ ?\ p(e^{-1}(r)) : 0]$
13:
14: **var** $\pi_1 := \pi_0 \cup \{ax_1, ax_2\}$ ▷ update path conditions
15:
16: **var** $h_{filtered} :=$ **under_approx**(h_0, p)
17: **try**
var $h_1 :=$ **remove**$(h_{filtered}, \pi_1, \lambda r.\ ch.p(r))$ ▷ remove greedily
18: **catch** (verification error)
var $h_1 :=$ **remove**$(h_0, \pi_1, \lambda r.\ ch.p(r))$ ▷ remove classically
19: **end try**
20:
21: **return** (h_1, π_1) ▷ return the updated heap and path conditions

4.3 Greedy read

So far, we have introduced a greedy approach for removing permissions. Now, we extend this concept to reading locations as well.

As stated in Section 2.2.3, Algorithm 4 merges all value mappings into a single new mapping, requiring an additional quantified constraint for each chunk. In programs with frequent read operations, this overhead can quickly accumulate, leading to performance issues.

However, reading operations, especially for single locations, are often fully constrained by a single chunk, avoiding the need to summarize the entire heap.

```
1  field f: Int
2  method foo() {
3      var x: Ref
4      var s1: Set[Ref], s2: Set[Ref]
5      inhale forall r: Ref ::
6          r in s1 ==> acc(r.f, write) && r.f == 1 // chunk 1
7      inhale forall r: Ref ::
8          r in s2 ==> acc(r.f, write) // chunk 2
9
10     assert forall r: Ref :: r in s1 ==> r.f == 1
11     assert x in s1 ==> x.f == 1
12 }
```

Listing 4.4: The reading operation in both assertions will result in heap summarization, even though it could be avoided since chunk 1 already entails all the required constraints.

As illustrated in Listing 4.4, there are scenarios where heap summarization is unnecessary, raising the question of whether it can be skipped. In many cases, it can, but in certain situations, doing so would lead to incompleteness. Specifically, if permissions to locations are split across multiple chunks, but only one chunk contains information about the values stored at these locations, skipping summarization could result in failing assertions.

Whether the assertion in Listing 4.5 succeeds or fails, depends on the order in which chunks are processed. If processing begins with chunk 1, the assertion will not succeed without a summarization as the value constraint is missing.

Nevertheless, we can attempt to greedily process the reading operation and fall back on summarization when necessary. This process works in the following way. At first, we attempt to find a single chunk that grants all required permissions. If we do not find one, we summarize the heap and retry. If we do find one, we either successfully complete the read operation or,

```
field f: Int
function c(r: Ref) : Bool
method foo() {
    var x: Ref
    inhale c(x)
    inhale forall r: Ref ::
        c(r) ==> acc(r.f, 1/2) // chunk 1
    inhale forall r: Ref ::
        c(r) ==> acc(r.f, 1/2) && r.f == 1 // chunk 2
    assert x.f == 1
}
```

Listing 4.5: Both chunks grant permissions for the same locations. However, only chunk 2 also contains information about the location's values.

if a verification error occurs due to missing value information, we summarize and retry.

Essentially, this approach trades quantified constraints in the path conditions for many small intermediate checks, such as verifying whether a chunk has sufficient permission. In general, this is a worthwhile trade-off because small checks, once completed, do not affect the rest of the verification process. In contrast, quantified constraints can, if unintentionally triggered, cause performance issues.

4.4 State consolidation

In the previous section, we encountered two different types of incompleteness when using the proposed greedy approaches. The first arises due to disjunctions, as seen in Listing 4.3, while the second occurs when multiple chunks represent permission to the same locations, e.g. Listing 4.5. While the first issue is difficult to resolve, the second can be addressed using state consolidation.

```
field f: Int
method foo() {
    var x: Ref
    inhale acc(x.f, 1/2) // chunk 1
    inhale acc(x.f, 1/2) // chunk 2
    // consolidated in new chunk
    assert acc(x.f, write) // fails without merging
}
```

Listing 4.6: Inhaling `write` permission for `x.f` is done in two steps resulting in two chunks that grant permission to the same location. To prove the assertion, these chunks must first be consolidated into a new chunk.

As described in Section 2.1.4, state consolidation refers to the process of merging two chunks if they grant permissions to the same locations. This step is necessary to address incompleteness issues such as the one demonstrated in Listing 4.6.

At the moment state consolidation is only implemented in Silicon for chunks that represent permission for a single location. However, the concept can be extended to quantified heap chunks. Let us define how two quantified chunks can be consolidated into a new quantified heap chunk.

$$ch_1 := [r.f \rightarrow v_1(r) \# p_1(r)] \tag{4.5}$$
$$ch_2 := [r.f \rightarrow v_2(r) \# p_2(r)] \tag{4.6}$$

The consolidation of chunk 4.5 and chunk 4.6 requires two steps. First, we need to sum up their permission and second, we need to summarize their value maps. As a result, we get a new chunk of the form of Equation 4.7.

$$\begin{aligned} ch_{new} := [r.f \rightarrow v_3(r) := summarize(\{\{ch_1, ch_2\}\})[0] \# \\ p_3(r) := p_1(r) + p_2(r)] \end{aligned} \tag{4.7}$$

Note that the resulting chunk has two negative side effects. First, the permission expression becomes more complex. Second, summarization adds quantified constraints to the path conditions, which might impact performance. However, consolidation improves completeness, as it prevents failures like the one in Listing 4.5. The main challenge lies in properly balancing this trade-off.

To decide when consolidation should occur, we first need a better understanding of the sources of incompleteness it can resolve. One primary source is heap fragmentation, which refers to the unintended splitting of chunks internally without any explicit intent from the user.

Consider the assignment operation `x.f := 1`. Internally, this operation is encoded as `exhale acc(x.f, write)` followed by `inhale acc(x.f, write)` $*$ `x.f = 1`. If the permission to `x.f` is granted by a quantified chunk, this operation splits the chunk in two, increasing the number of chunks and potentially leading to incompleteness, as illustrated in Listing 4.7.

A similar issue arises when using `unfold` and `fold` statements, which can also contribute to heap fragmentation.

In Listing 4.7, an interesting phenomenon occurs. We can think about the newly created chunks as sharing the same parent chunk (the original chunk before the fragmentation). Conceptually, these heap modifications can be represented by a directed acyclic graph in which each chunk that spawned

```
1 field f: Int
2 method foo() {
3     var x: Ref, y: Ref
4     var s: Set[Ref]
5     inhale x in s && y in s
6     inhale forall r: Ref :: r in s ==> acc(r.f, write)
7     x.f := 1 // fragments the heap into two chunks
8     assert acc(y.f, write) // fails without merging
9 }
```

Listing 4.7: An example of an incompleteness issue in the greedy approach caused by heap fragmentation.

from a different chunk is considered its child. The leaves of this graph represent the current heap. Formally, the overall structure is a forest, because some chunks might not have a parent, but each connected component of the forest forms a tree.

Within any such tree, all leaves have originated from the same initial ISC. The reason for splitting them into multiple chunks is driven by internal implementation details rather than by explicit user intent. Consequently, there is a strong likelihood that the user still intends to treat these chunks as a single unit. This observation provides the basis of our chunk-merging strategies. All leaves of a tree are consolidated into the same chunk, restoring a similar version to the original chunk.

Three of our four consolidation strategies presented next follow this principle. The fourth strategy is more aggressive as it always merges all chunks.

4.4.1 Tag system

The first strategy, which we will refer to as the tag system, extends each chunk with an additional identifier. If a chunk is a child of another chunk in the described graph representation, it inherits the tag from its parents. As a result, all leaf chunks within a tree are consolidated together, adhering to the previously described principle.

$$ch_1 := [r.f \rightarrow v_1(r) \# p_1(r) \# \boldsymbol{tag(1)}] \tag{4.8}$$

Note that it is possible that chunks have multiple parents. In such cases, the inherited tag is the most frequently occurring one among all parent chunks.

The major advantage of this approach is that consolidating does not require the SMT solver. Since tag comparisons can be done within Silicon, making this strategy very fast.

However, this comes at a cost. The strategy can only allow chunks to inherit from each other if the fragmentation occurs within the encoding.

```
1  field f: Int
2  method foo() {
3      var x: Ref, y: Ref
4      var s: Set[Ref]
5      inhale x in s && y in s
6      inhale forall r: Ref :: r in s ==> acc(r.f, write)
7      exhale acc(x.f, write) // manual fragmentation of heap
8      inhale acc(x.f, write) && x.f == 1
9      assert acc(y.f, write) // fails without merging
10 }
```

Listing 4.8: An example of an incompleteness issue in the greedy approach caused by manual heap fragmentation.

Even though Listing 4.8 is semantically equivalent to Listing 4.7, the tag system is unable to consolidate the two chunks. The reason is that the fragmentation is now explicit rather than being handled internally. This situation arises, for example, in Prusti [6] one of Viper's frontends, meaning that the fragmentation is still not explicitly intended by the user. To address such issues, the next strategy proves useful.

4.4.2 Single receiver

The goal of this strategy is to ensure that if permissions for a single location are split off from one chunk into a new chunk, they can be merged back into a single chunk. To achieve this, the quantified chunk retains information about the single locations for which it lost permissions.

$$ch_1 := [r.f \rightarrow v_1(r) \# p_1(r) \# \boldsymbol{locations}(x)] \tag{4.9}$$
$$ch_2 := [x.f \rightarrow v_2 \# 1] \tag{4.10}$$

Consolidation can occur through syntactic equality between the remembered location and the receiver of another chunk or by leveraging aliasing information with the help of the SMT solver.

The major advantage of this strategy is that it enables more merging compared to the tag system, thus improving overall completeness. The main disadvantage is that it requires assistance from the SMT solver. However, this is limited to simple and fast checks, making the overhead minimal.

4.4.3 Original condition

The previous strategy can be extended to track all lost permissions to locations, rather than just a single location. This is especially useful for method calls where ISCs are commonly used as preconditions and postconditions. In many cases, it is beneficial to merge the postconditions back into the original chunk from which the preconditions were removed.

```
1  field f: Int
2  method bar(s: Set[Ref])
3     requires forall r: Ref :: r in s ==> acc(r.f, write)
4     ensures forall r: Ref :: r in s ==> acc(r.f, write)
5
6  method foo() {
7      var s1: Set[Ref], s2: Set[Ref]
8      var x: Ref
9      inhale s2 subset s1
10     inhale x in s1
11     inhale forall r: Ref :: r in s1 ==> acc(r.f, write)
12     bar(s2) // fragmented heap
13     assert acc(x.f, write) // fails without merging
14 }
```

Listing 4.9: An example of an incompleteness issue in the greedy approach caused by manual heap fragmentation of quantified chunks.

The method call in Listing 4.9 causes multiple locations to be split off from the original ISC on line 12, resulting in heap fragmentation. The previous strategy fails to merge these fragmented chunks, highlighting the need to track multiple locations rather than just a single one.

Conceptually, there are two possible approaches to achieving this. One option is to track all lost locations, but this can be challenging, especially when locations are removed across multiple operations. A simpler and more effective approach is to store the original condition function instead. Removed permissions always form a subset to the locations where the original condition holds. As a result, checking whether chunks should be merged becomes much simpler.

$$ch_1 := [r.f \rightarrow v_1(r) \# p_1(r) \# \boldsymbol{c_{org}(r) := (r \textit{ in } s1)}] \tag{4.11}$$

Equation 4.11 shows the original condition that would be stored in the chunk created by the `inhale` statement on line 12 in Listing 4.9.

The main advantage of this strategy is its ability to merge chunks across a broad range of scenarios. However, this comes at the cost of more complex checks, introducing additional overhead for the SMT solver.

4.4.4 Full merging

This is the only strategy that does not follow the previously explained principle for determining when to merge chunks. This strategy always merges all chunks, theoretically yielding a complete algorithm. Since there is always at most one chunk, the greedy approach will pick the chunk containing all available permissions.

However, maximizing merging comes at the cost of generating complex permission terms, which diminishes the primary advantage of the proposed greedy approach.

Moreover, implementing this strategy without introducing incompleteness proves to be challenging in practice. Further details on these incompletneses can be found in Section 4.6.5. Despite its high likelihood of causing performance issues, this strategy is still included for comparative purposes.

4.4.5 Final remarks

The fourth strategy, which merges all chunks, differs significantly from the other three strategies, as it does not follow the previously established principles for determining when to merge chunks.

In contrast, the first three strategies operate independently, making them combinable. Each successive strategy increases the potential for merging chunks but comes at the cost of higher computational complexity.

Ultimately, the choice of strategy depends on balancing completeness and performance. One example of combining them is to apply them sequentially, where each strategy is attempted only if the previous one fails.

4.5 Implementation

The following section details the implementation of the previously discussed approach, focusing primarily on differences between the conceptual explanation and its practical implementation. Although the approach is also implemented for predicates and magic wands, they operate on the same underlying principle. For simplicity, the implementation is described only for fields.

4.5.1 Greedy exhale

The greedy exhale mechanism is largely implemented as shown in Algorithm 7, with two key differences.

First, instead of relying on a `try-catch` block, Silicon employs a more advanced retry mechanism, which automatically performs state consolidation before retrying.

Previously, this was not applied to quantified chunks because the over-approximation algorithm did not benefit from state consolidation. However, since the under-approximation approach does, all calls to Algorithm 7 are now wrapped within this retrying mechanism to improve completeness.

Second, the resulting heap's chunks are categorized into three groups: chunks that no longer hold permissions for any location, chunks that still hold some permissions but have changed, and chunks that remain unchanged.

If the total number of chunks in the first two categories is exactly one, then no value-map summarization is required. This optimization further reduces the number of quantified axioms, but it also introduces the same incompleteness issues as the greedy read approach, as both techniques follow the same principle.

Lastly, since this approach requires SMT solver checks, it raises the question of what timeout should be used. By default, Silicon provides a 10 millisecond timeout, which can be adjusted by the user using the `--checkTimeout <val>` flag. However, this default value is often too short for this type of check, so it is multiplied by a constant factor. Although the optimal factor varies depending on the example, in practice a factor of 10 has been found to work well.

4.5.2 Greedy read

Whenever a field value is read, such as `x.f`, the greedy read implementation iterates over all chunks and checks whether $ch_i.p(x) > 0$ holds. If the check succeeds, the corresponding chunk's value for that reference, $ch_i.v(x)$, is returned. If no chunk satisfies the condition, Silicon falls back to executing Algorithm 4, as it has done previously.

The key question is again how much time should be spent on checking $ch_i.p(x) > 0$. Since the default 10 millisecond timeout is often insufficient, a constant factor of 10 is applied.

4.5.3 Switching algorithms

Since there are now two algorithms for handling quantified chunks, it is logical to allow users to switch between them. To achieve this, an existing

flag in Silicon is repurposed.

Silicon already provides the command-line flag `--exhaleMode <val>`, which allows switching between different exhaling algorithms for single locations. This functionality has now been extended to also apply to quantified locations.

- `--exhaleMode 0`: Only the greedy approach is used.
- `--exhaleMode 1`: Only the over-approximating algorithm is used.
- `--exhaleMode 2`: First, the greedy approach is used. If it fails, it is retried with the over-approximating algorithm.

Internally, Silicon selects between algorithms by modifying the proof obligations that determine whether a chunk is relevant.

- **Mode 0** always uses the checks from Equations 4.4 and 4.3, using only the greedy algorithm .
- **Mode 1** sets the proof obligation to `true`, effectively considering all chunks relevant and subsequently only uses the over-approximating algorithm.
- **Mode 2** starts with the greedy checks, and if they fail, the retry mechanism updates to mode 1, thus switching to the over-approximating algorithm.

4.5.4 State consolidation

This section covers the intricacies of implementing state consolidation for quantified chunks.

Singleton quantified chunks

Internally, Silicon distinguishes between three types of chunks when creating them: non-quantified chunks, quantified chunks, and singleton quantified chunks. Non-quantified chunks describe a value and permission for a single location. Quantified chunks extend this concept by describing values and permissions for multiple locations. Singleton quantified chunks, on the other hand, are a special case of quantified chunks that represent only one location, making them a hybrid between quantified and non-quantified chunks.

Silicon statically determines for each field definition whether its usage should be represented using quantified chunks or non-quantified chunks. This distinction is necessary for technical reasons, which are discussed in Schwerhoff's PhD thesis [18] and will not be explored further here.

Although this distinction is made for practical reasons, in theory, non-quantified chunks are not necessary, as any chunk could be defined as a quantified chunk. The following definitions illustrate this concept.

$$ch_{single} := [x.f \rightarrow v \# p] \tag{4.12}$$

$$ch_{quantified} := [r.f \rightarrow r = x \ ? \ v : v_1(r) \# r = x \ ? \ p : 0] \tag{4.13}$$

The chunks ch_{single} and $ch_{quantified}$ are semantically identical but are represented differently, with one as a non-quantified chunk and the other as a quantified chunk.

However, using non-quantified chunks offers a major advantage. Their definitions do not require any quantified axioms to be added to the path conditions, such as those for inverse functions. As a result, the underlying SMT can reason about it more efficiently.

For performance reasons, it is therefore beneficial to treat as many chunks as possible as non-quantified chunks, while still maintaining the distinction for each field definition.

To avoid unnecessary quantified chunks, singleton quantified chunks are introduced. They are still defined like quantified chunks, but additionally store a reference that allows them to be instantiated, enabling them to be treated as non-quantified chunks whenever they are used.

$$ch_{singleton} := [r.f \rightarrow r = x \ ? \ v : v_1(r) \# r = x \ ? \ p : 0 \# \{x\}] \tag{4.14}$$

The chunk in Equation 4.14 stores the reference `x`. By instantiating this reference, the chunk is transformed into a non-quantified chunk, eliminating the need to define an inverse function.

```
1  field f: Int
2  field g: Int
3  function c(r: Ref) : Bool
4  method foo() {
5      var x: Ref
6      inhale c(x)
7      inhale forall r: Ref ::
8          c(r) ==> acc(r.f, 1/2) // quantified chunk
9      inhale acc(x.g, 1 / 2) // non-quantified chunk
10     inhale acc(x.f, 1 / 2) // singleton quantified chunk
11 }
```

Listing 4.10: Illustrates how different `inhale` statements generate different chunk types in Silicon.

The example in Listing 4.10 demonstrates how multiple fields are translated into different chunk types based on their usage in `inhale` statements. Since field `f` appears in an ISC, all corresponding chunks are quantified chunks. The statement `inhale acc(x.g, 1 / 2)` results in a singleton quantified chunk that also stores the reference `x`. Field `g` is never used in an ISC. As a result, its permissions can be represented using a non-quantified chunk.

This distinction between chunk types complicates state consolidation and can lead to incompleteness if not handled carefully.

Whenever singleton quantified chunks are merged with quantified chunks, the result is a new quantified chunk, causing the reference information from the singleton chunk to be lost. This loss introduces incompleteness issues.

```
1  field f: Int
2  function c(r: Ref) : Bool
3
4  method foo() {
5    var x: Ref
6
7    inhale forall r: Ref :: {r.f} x == r ==> acc(r.f, 1 / 2)
8    inhale acc(x.f, 1 / 2) // needed for triggering
9    inhale forall r: Ref :: {r.f} c(r) ==> perm(r.f) == none
10
11   assert !c(x)
12 }
```

Listing 4.11: Illustrates the interaction between quantified chunks and singleton quantified chunks. Verification succeeds only because the singleton quantified chunk triggers the quantified constraint on line 9.

In the example in Listing 4.11, the quantified constraint on line 9 must be triggered in order to prove the final assertion. Since a singleton quantified chunk exists, it is instantiated as `x.f` and directly passed to the SMT solver, rather than being defined using quantified axioms.

This instantiation, in turn, triggers the quantified constrained, as `x.f` matches the trigger `r.f`. Consequently, the necessary conditions are established to successfully prove the assertion.

If the singleton quantified chunk is not instantiated, for example, if it is merged into another chunk through state consolidation, the verification would fail, leading to a loss of completeness.

This example illustrates how naively applied state consolidation can introduce incompleteness issues. However, it is still possible to merge these chunks in a way that preserves completeness. This is achieved by tracking all singleton

instantiations across all chunks, allowing them to be instantiated later when needed.

$$ch_1 := [r.f \rightarrow v_1(r) \# p_1(r)] \tag{4.15}$$
$$ch_2 := [r.f \rightarrow r = x \ ? \ v_x : v_2(r) \# r = x \ ? \ p_x : 0 \# \{x\}] \tag{4.16}$$
$$ch_3 := [r.f \rightarrow r = x \ ? \ v_y : v_3(r) \# r = y \ ? \ p_y : 0 \# \{y\}] \tag{4.17}$$
$$ch_{merged} := [r.f \rightarrow v_4(r) \# p_4(r) \# \boldsymbol{\{x, y\}}] \tag{4.18}$$

Equation 4.18 shows how chunks ch_1, ch_2 and ch_3 are merged while preserving all singleton instantiations. Whenever ch_{merged} is used, it is treated as normal quantified chunk, and additionally, all stored references are instantiated. This ensures that no incompleteness issues arise due to merging.

Triggers

Another challenge with merging quantified chunks involves inverse functions. When a user does not specify a trigger, Silicon attempts to infer one automatically. One such inferred trigger is the inverse function of a chunk.

These triggers are then used to define internal axioms, also referred to as properties. Properties have already been discussed in detail in Section 3.3.3. The key insight here is that if a property is a quantified axiom, it requires a trigger. This trigger, in turn, depends on the chunk's inverse function.

As a result, a consolidated chunk must retain all triggers from the original chunks from which it was derived, including all inverse functions. Implementing this correctly has proven to be particularly challenging.

4.5.5 Merging strategies

This section provides implementation details for the four proposed merging strategies.

Full merging

We begin with the implementation of the strategy where all chunks are merged together. This approach does not require any extensions to the quantified chunk definition, nor does it introduce additional checks for the SMT solver. Additionally, it eliminates the need to filter which chunks should be merged, making it a straightforward implementation with no significant complexities.

Tag system

Next, we discuss the implementation of the tag system. The implementation is mostly straightforward, requiring only the addition of a tag to each chunk.

However, an important consideration is that every chunk, including non-quantified chunks, must be assigned a tag.

```
1  field f: Int
2  predicate P(r: Ref) {
3      acc(r.f, write)
4  }
5
6  method foo() {
7      var x: Ref
8      var s: Set[Ref]
9      inhale x in s
10
11     inhale forall r: Ref :: {P(r)} r in s ==>
12         acc(P(r), write) // quantified chunk with tag(i)
13     unfold P(x)
14     assert acc(x.f, write) // non-quantified chunk tag(i)
15     fold P(x) // singleton quantified chunk with tag(i)
16 }
```

Listing 4.12: Illustrates the necessity of passing tags between quantified and non-quantified chunks to enable merging after `unfold` and `fold`.

The example in Listing 4.12 demonstrates how the `unfold` operation requires passing the tag between quantified and non-quantified chunks. Without this mechanism, the singleton quantified chunk created by the `fold` operation would store a new tag, preventing it from being merged back with its original chunk. As a consequence, this would greatly limit the usefulness of the tag system.

Finally, Silicon must store the current tag that needs to be passed to new chunks. This is achieved by adding it to its internal state, which maintains various relevant data for the verification process. This tag is then updated throughout the verification process.

Single receiver

Next, we discuss the single receiver strategy, which is simpler to implement as it only introduces modification to quantified chunks. Given a chunk, such as Equation 4.7, and a list of singleton quantified chunks, it is necessary to iterate over the list with each singleton reference stored in the chunk, so in this case, `x` and `y`. The chunks are merged if and only if the references are either syntactic equivalent or the SMT solver proves that they are aliases.

Since this strategy may require checks performed by the SMT solver, the

timeout for these checks is set to 10 milliseconds by default and can be adjusted by the user using the flag `--checkTimeout <val>`, where `<val>` is specified in milliseconds. As this check is efficient and incurs minimal computational overhead, the default timeout is typically sufficient. Further details about this timeout are found in Section 4.6.5.

Original condition

Lastly, we discuss the original condition strategy, which is implemented similarly to the single receiver strategy. The key difference is that, rather than storing individual references, each chunk preserves the original condition from its initial creation. Additionally, an SMT solver check is always required to determine whether two chunks should be merged.

$$\forall r : Ref.\ ch_1.p(r) > 0 \implies ch_2.c_{org}(r) \tag{4.19}$$
$$\forall r : Ref.\ ch_2.p(r) > 0 \implies ch_1.c_{org}(r) \tag{4.20}$$

The checks work similarly to the greedy approach. They determine whether the locations for which one chunk grants permission form a subset of the locations where the original condition of the other chunk holds. If this is true, the chunks should be merged.

An important detail is that these checks are more complex than those in the previous strategy because they contain quantifiers. For this reason, the default check timeout of 10 milliseconds is often too short and is therefore multiplied by a constant factor.

The optimal value for this factor varies depending on the example, but in practice a factor of 10 has been found to work well. Further details about this timeout can be found in Section 4.6.5.

4.6 Evaluation

In this section we evaluate the proposed greedy approach. The primary goal is to demonstrate that it is less dependent on chunk order heuristics and to compare its performance to the over-approximating algorithm. We also investigate potential completeness issues and examine the proposed state consolidation strategy. Finally, the approach is applied to parts of VerifiedSCION [16] and the resulting verification behavior is analyzed.

4.6.1 Increasing complexity examples

Understanding and testing Silicon's verification behavior requires examples where the verification effort can be scaled. We refer to these as increasing

complexity examples. The following four programs cover a range of scenarios commonly encountered when using ISCs. Each example is designed to allow controlled increases in verification effort, enabling a meaningful comparison between the greedy and the over-approximating algorithm.

Next, we introduce each example by first explaining the scenario it represents and then describing how its complexity can be systematically increased.

ManySets

The purpose of this program is to demonstrate a situation where Silicon's heuristics do not pick an efficient chunk order. We introduce multiple sets, each represented by its own chunk in the verification process. With a final assignment, Silicon is required to find the right chunk, which is done in a bad chunk processing order and results in a decrease of verification time.

```
1  method manySets() {
2      var a0: Set[Ref]
3      var a1: Set[Ref]
4      inhale forall a: Ref :: a in a0 ==> acc(a.f, write)
5      inhale forall a: Ref :: a in a1 ==> acc(a.f, write)
6      var r: Ref
7      var i: Int
8      inhale r in a1
9      r.f := i
10 }
```

Listing 4.13: The assignment to `r.f` requires full permission, causing Silicon to process all chunks in a suboptimal order.

Listing 4.13 illustrates the idea with two sets. The complexity can be scaled up by adding more sets and corresponding `inhale` operations.

ArraySplit

This example demonstrates how a single ISC can be split into multiple ISCs with different conditions and later be recombined into one. It requires several checks in the greedy algorithm, making it well-suited for evaluating whether the overhead introduced by these checks impacts performance.

Listing 4.14 presents a simplified version with two splits, omitting pre- and postconditions for clarity. The complexity of the example can be increased by dividing the array into more loop invariants, which also raises the number of read operations.

```
method arraySplit(a0: IArray) returns (res:IArray) {
    var i: Int := 0
    while(i < 5 )
        invariant  0 <= i && i <= 5
        invariant len(a0) == 10
        invariant forall j: Int :: {loc(a0, j)}
            0  <= j && j < 5 ==> acc(loc(a0, j).val, 2/3)
        invariant forall j: Int :: {loc(a0, j)}
            5  <= j && j < 10 ==> acc(loc(a0, j).val, 2/3)
        invariant forall j: Int :: {loc(res, j)}
            0 <= j && j < 5 ==> acc(loc(res, j).val, write)
    {
        loc(res, i).val :=
            loc(a0, i+0).val + loc(a0, i+5).val
        i := i + 1
    }
}
```

Listing 4.14: Splitting a single ISC into multiple ISCs with different conditions. Pre- and postconditions are omitted for simplicity.

ParallelArray

This program illustrates how multiple `exhale` operations can lead to a complex permission expression, testing whether the newly added checks in the greedy approach are sufficient to complete the proof or whether they result in a timeout. The program takes an array and two values and updates all elements equal to the first value by replacing them with the second. This behavior is encoded in a parallel fashion.

Initially, all permissions are represented by a single chunk, which is gradually exhaled in multiple steps. A key challenge is that Silicon must verify that all permissions have been removed from the original chunk after the last `exhale`. If this check fails, the chunk is not deleted, which causes instability.

Listing 4.15 presents a simplified version with two parallel branches. For clarity, the base case and the pre-/postconditions are omitted. The complexity of the example can be increased by adding more parallel branches, i.e. dividing the array into more segments and introducing additional `exhale` and `inhale` operations.

ArraySum

This example was previously introduced in Section 4.1 and is shown in Listing 4.1. That section also explains how increasing the number of arrays raises the verification effort for this program. It was chosen because it involves

```
method parallelArray(a: IArray, left: Int,
    right: Int, from: Int, to: Int) {
    if (right - left <= 1) {
        // base case
    } else {
         var mid: Int := left + (right - left)  /  2
        exhale 0 <= left && left < mid && mid <= len(a)
        exhale 0 <= mid && mid < right && right <= len(a)
        exhale forall i: Int ::
            left <= i && i < mid ==> acc(loc(a, i).val)
        exhale forall i: Int ::
            mid <= i && i < right ==> acc(loc(a, i).val)
        inhale 0 <= left && left < mid && mid <= len(a)
        inhale 0 <= mid && mid < right && right <= len(a)
        inhale forall i: Int ::
            left <= i && i < mid ==> acc(loc(a, i).val)
        inhale forall i: Int ::
            mid <= i && i < right ==> acc(loc(a, i).val)
    }
}
```

Listing 4.15: A simplified parallel encoding of updating values in an array.

many similar ISCs and numerous read operations, making it an interesting case for observing how the greedy approach behaves during verification.

4.6.2 Experimental setup

All experiments were carried out on the Euler cluster at ETH Zurich [2], using a single CPU and 8 GB of RAM. Each benchmark was verified 30 times using the Silicon backend, and the results were averaged over all runs. During these runs, we collected various metrics from both Silicon and the underlying SMT solver Z3.

Silicon metrics

Our primary metric is execution time, although it can vary and may not always reflect improvements in non-bottleneck areas. To gain deeper insights, we also monitor the number of SMT checks, since our approach introduces additional checks. Note that a higher number of checks is not inherently problematic as long as overall verification time is not affected.

Z3 metrics

We also track some Z3 statistics, focusing primarily on the number of quantifier instantiations, but also considering the number of conflicts. A conflict occurs when the solver encounters a contradiction while exploring the search space. Typically, a high number of conflicts indicates complex constraints, making this metric useful for understanding how difficult a problem was for Z3 to solve.

4.6.3 Greedy exhale

We begin by evaluating the dependency of our greedy approach on the chunk processing order, in comparison to the over-approximating approach. To this end, we execute the previously introduced examples with increasing complexity, disabling Silicon's chunk order heuristics using the flag `--disableChunkOrderHeuristics`.

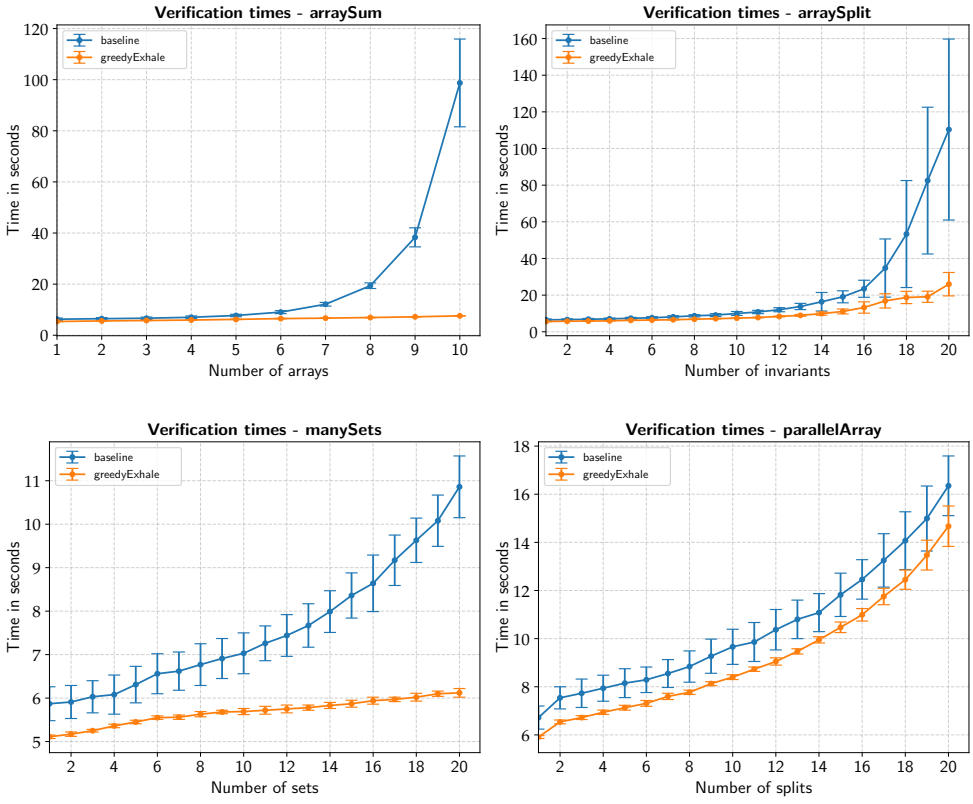

Figure 4.3: Verification times for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows verification times in seconds. Data points are averaged, and the standard deviation is included to indicate stability.

We can observe in Figure 4.3 that verification time improves across all examples. Although the improvement for `parallelArray` is minimal, the other examples show exponential speedups. Notably, the graph for `arraySum` stops at 10 arrays, as verifying larger instances becomes too time consuming. Both `arraySplit` and `manySets` also exhibit exponential improvements, though not as pronounced as `arraySum`. It is worth highlighting that verification stability improves in all cases.

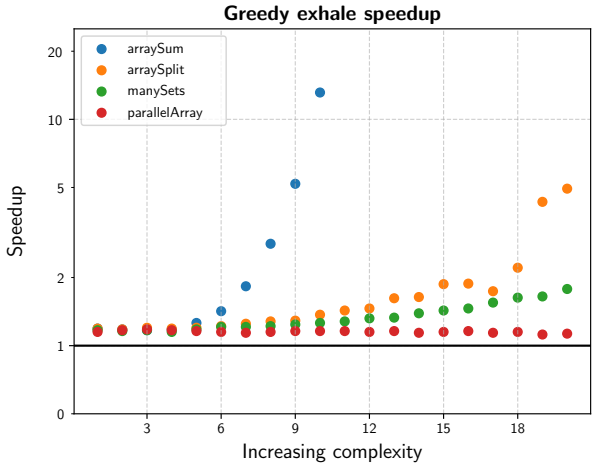

Figure 4.4: Speedup plot of examples. The X-axis represents the complexity level, while the Y-axis indicates the speedup. Data points represent the median verification time. The black line serves as the baseline reference at speedup $= 1$.

Figure 4.4 shows how the speedup scales with increasing complexity. We observe that the speedup for all examples are above one. While `parallelArray` only exhibit a minor constant improvement, `arraySum` and `arraySplit` stand out with significant speedups of over 10 and over 5, respectively.

Next, we evaluate the number of quantifier instantiations. In three of the four examples in Figure 4.5, we observe a reduction in quantifier instantiations. The exception is `parallelArray`, which shows a slight increase due to the additional checks introduced by the verification process.

Although we do not include explicitly graphs here, it is worth noting that while `arraySum` benefits significantly from enabling Silicon's chunk order heuristic, the other three examples are largely unaffected by it. As a result, the speedups observed in `arraySplit` and `manySets` are still achieved even with the heuristic enabled.

Next, we evaluate our approach in combination with an optimization called conditionalize permissions. A very high level summary is that condition functions and permission functions are combined in order to reduce the number of branches Silicon must explore during verification. In practice, this

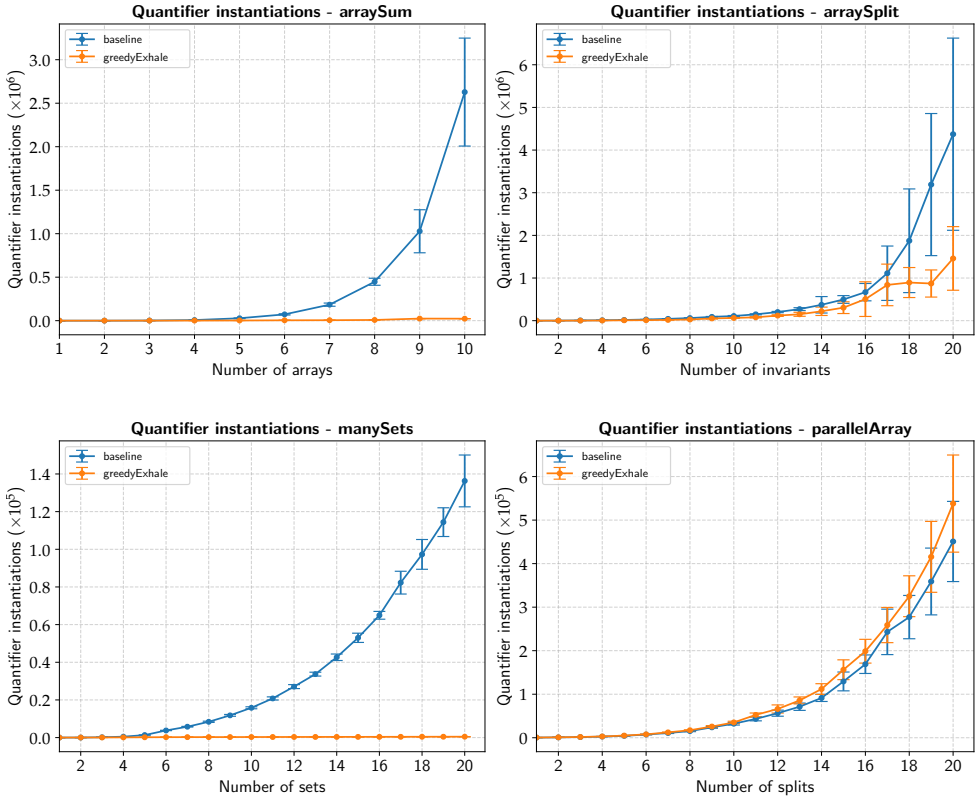

Figure 4.5: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

has proven to be highly effective. However, it can also interfere with chunk processing order, potentially causing performance issues with ISCs. Our goal is to assess whether this problem persists in the greedy approach. To do so, we evalute the examples with both conditionalize permissions and chunk order heuristics enabled.

As shown in Figure 4.6, the greedy approach is both faster and more stable in three out of the four examples. The only exception is `arraySum`, which is unaffected by conditionalize permissions and performs similarly as the greedy approach.

Overall, these results demonstrate that the new greedy approach is less sensitive to the chunk order heuristic and integrates more effectively with the conditionalize permissions optimization. We evaluate the completeness of this algorithm in combination with the state consolidation strategies at the end of this chapter.

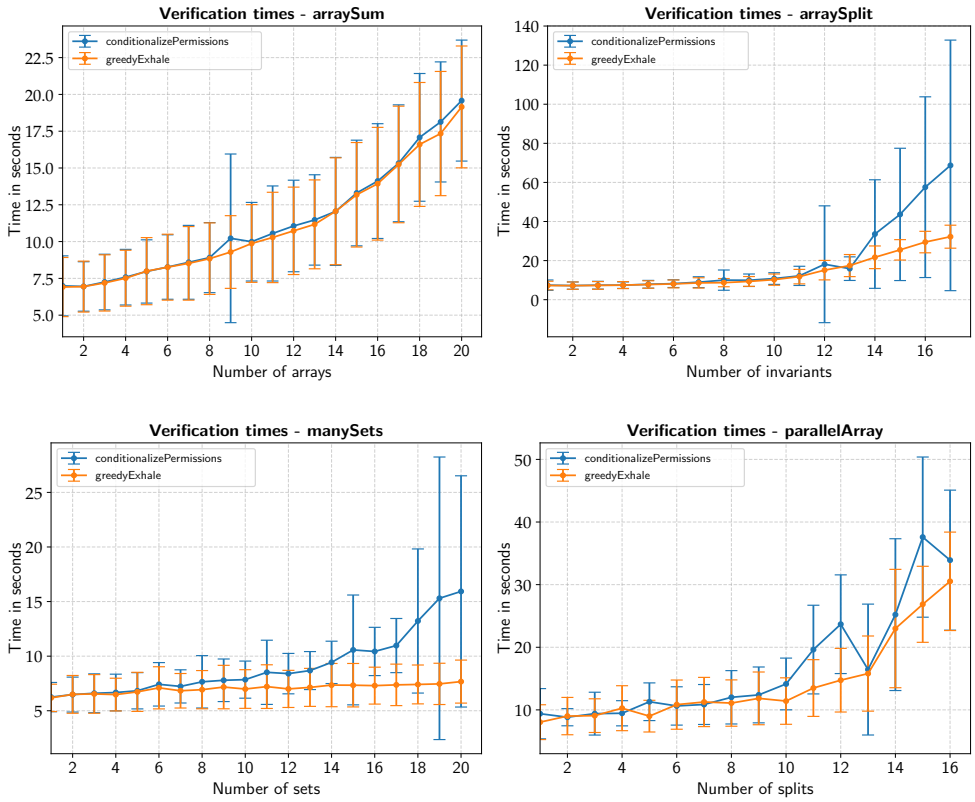

Figure 4.6: Verification times for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows verification times in seconds. Data points are averaged, and the standard deviation is included to indicate stability.

GitHub issue

As a final part of this evaluation, we pick a performance related GitHub issue [11] filed in summer 2024. It involves automatically generated Viper programs containing many arrays. Verification of those programs leads to performance issues. A related pull request [20] proposes a fix by merging identical quantified chunks. While this approach improves performance in many scenarios, we aim to show that it does not resolve the underlying problem in this particular case.

The root cause stems again from the interaction between the chunk order heuristics and the conditionalize permissions optimization. The following graphs illustrate the issue.

As shown in Figure 4.7 the combination of the chunk order heuristic with conditionalize permissions increases verification time exponentially. The greedy approach, however, matches the performance of the over-approximating algorithm, allowing users to continue benefiting from conditionalize permissions.

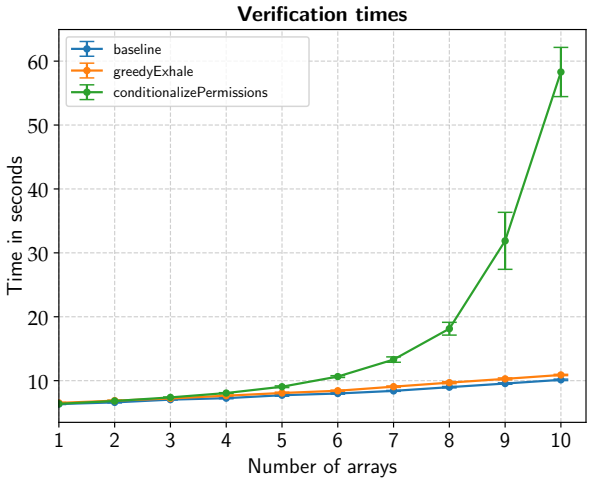

Figure 4.7: Verification times for examples from GitHub issue [11]. The X-axis represents the number of arrays, while the Y-axis shows verification times in seconds. Data points are averaged, and the standard deviation is included to indicate stability.

4.6.4 Greedy read

We begin by evaluating the number of quantifier instantiations with the known examples with complexity increases. However, we leave out `manySets` as this example does not contain any read operations.

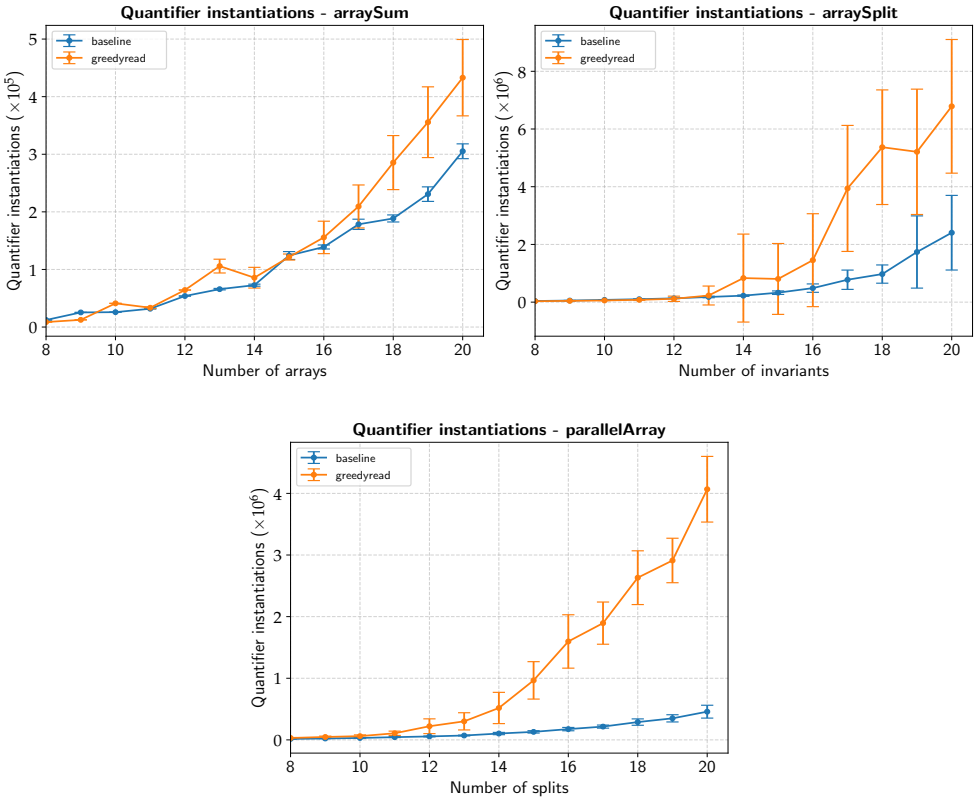

Figure 4.8: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

Figure 4.8 shows that in all examples, the number of quantifier instantiations increases. This suggests that for smaller examples the overhead from finding a single chunk with sufficient permissions outweighs the benefits gained from avoiding summarizations. To confirm this suspicion, we examine the total number of SMT checks performed.

We observe in Figure 4.9 that all examples show an increase in the number of SMT checks, with `parallelArray` being an extreme case. This highlights a core issue with this approach. As the number of chunks increases, so does the number of checks to find a single chunk that provides sufficient

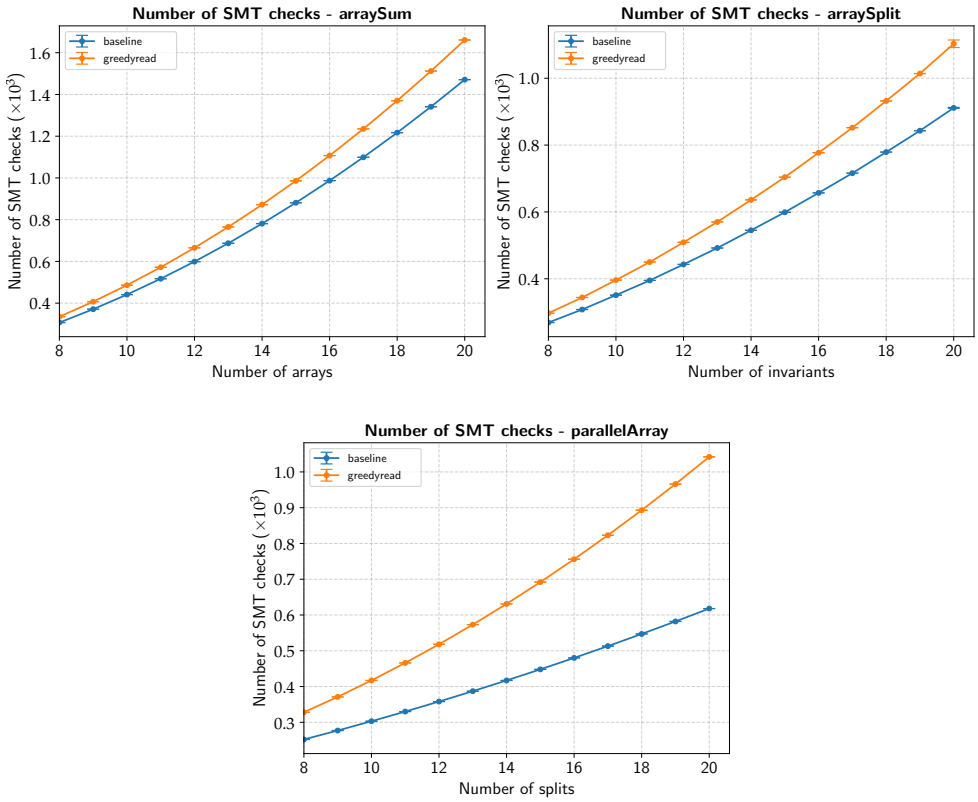

Figure 4.9: Number of SMT checks for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of SMT checks. Data points are averaged, and the standard deviation is included to indicate stability.

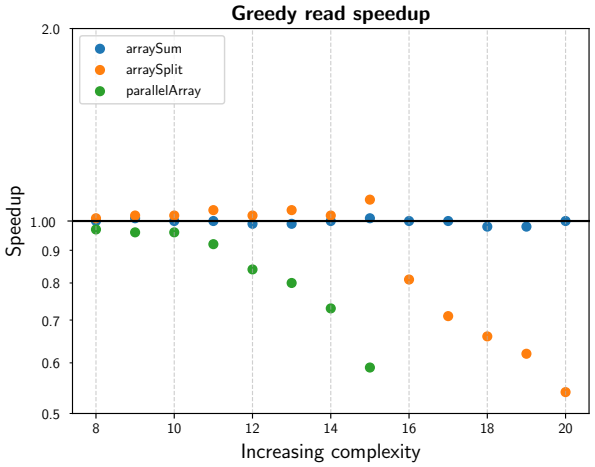

Figure 4.10: Speedup plot of examples. The X-axis represents the complexity level, while the Y-axis indicates the speedup. Data points represent the median verification time. The black line serves as the baseline reference at speedup = 1.

permissions for a read operation. Even with short timeouts, the overhead from these numerous checks accumulates and negatively impacts overall verification time. Figure 4.10 shows that, at best, the speedup remains around one. However, as the number of chunks increases, performance drops significantly.

Overall, this optimization does not perform as hoped. It depends heavily on the order in which chunks are checked which we were trying to avoid.

However, there is a silver lining. Preliminary tests using a more advanced chunk selection strategy based on syntactic similarity have shown promise. That said, implementing such a general strategy would require substantial effort.

VerifiedSCION

On VerifiedSCION, the evaluation showed more promise. For smaller methods involving fewer quantified chunks in the `slayers` and `scion` packages, the results were encouraging. In these cases, the number of quantifier instantiations was reduced by approximately a third and a fifth, respectively. This suggests that for methods with a limited number of quantified chunks but frequent read operations, the optimization can improve overall performance. We did not measure total verification time, as the large size of these packages introduces significant variability, making it difficult to obtain meaningful performance measurements.

Timeouts

Determining an ideal timeout is difficult as it depends on both the machine and the specific example. For simple cases, such as those seen earlier, a timeout of 10 milliseconds is more than sufficient since Z3 can solve those proof obligations almost instantly. However, in more complex programs, especially those generated by frontends, this is often not enough.

In VerifiedSCION, for instance, a 10 ms timeout frequently results in failed checks that Z3 could prove with a timeout of 100 ms. However, it is worth noting that on particularly fast machines a 10 ms timeout may still suffice as Z3 could complete all checks within that limit. Importantly, a higher timeout does not mean Z3 always uses the full duration. Most checks still finish within 10 – 30 ms, though a few may take longer. While higher timeouts introduce slight overhead, lower timeouts lead to many unnecessary failures.

We therefore recommend using a timeout of 100 milliseconds as a practical default but also note that this should likely be reduced on very fast machines.

4.6.5 State consolidation

First, we investigate our suspicion that the full merging strategy may lead to performance issues. To test this, we verify the `arraySum` example using the greedy approach with different timeout settings for the chunk selection check.

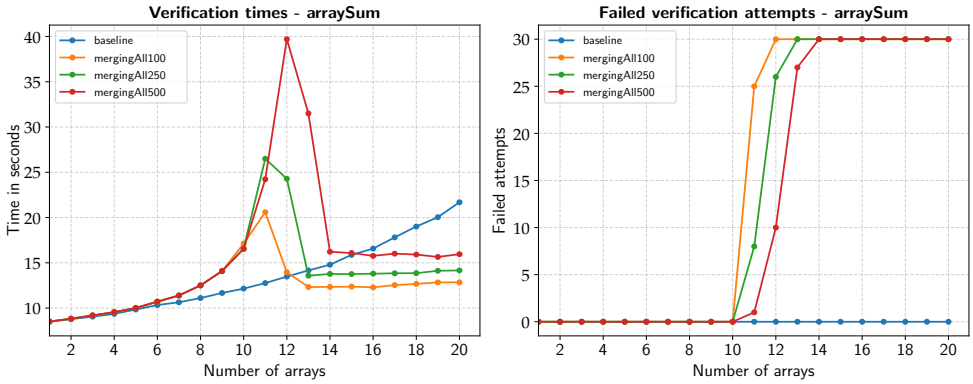

Figure 4.11: Verification times (left) and number of failed verification attempts (right) for `arraySum`. The X-axis represents the number of arrays, while the Y-axis shows verification time in seconds (left) and the number of failed attempts (right). On the left, data points are averaged, but standard deviation is omitted, as it does not provide meaningful insight when verification may fail. Both graphs compare the greedy approach combined with full merging, evaluated under three different timeouts for the chunk selection check: 100 ms, 250 ms, and 500 ms.

In the left graph in Figure 4.11, we observe a sharp increase in verification time as the number of arrays grows. This continues until the permission expression becomes so complex due to excessive merging that verification fails entirely because no chunk can be picked in time. At that point, verification times drop dramatically, as the process terminates quickly upon failure.

The right graph shows how many of the 30 verification attempts failed for each configuration with each number of arrays. It highlights that even a massive increase in timeout, from 100 ms to 500 ms, only delays failure by one or two additional arrays. The graph also illustrates how quickly verification becomes unstable before it ultimately fails completely.

One final observation is that, although verification time grows exponentially with excessive merging, it still scales better than the over-approximating algorithm without chunk order heuristics. This is because the complexity of the permission expression increases linearly with each chunk, not exponentially.

In summary, merging chunks excessively is rarely beneficial. The complexity of the resulting permission function in the merged chunk quickly leads to verification failure.

Completeness

In this section we evaluate the completeness of the greedy exhale approach combined with the proposed merging strategies. For that purpose, we ran Silicon's test suite with the greedy algorithm and analyzed the result. Overall, we encountered 18 verification errors.

We were able to resolve 3 out of the 18 errors by increasing the timeout of the chunk selecting check from original 10 ms to 100 ms. This larger timeout worked the best without incurring noticeable overhead. Hence, we recommend this as a default. Note that the timeout heavily depends on the used machine.

Furthermore, we identified 6 out of the 18 tests as being affected by heap fragmentation. We aim to investigate these cases in more detail to determine which chunk strategy can resolve this incompleteness.

Examples	Greedy exhale	Tag system	Single receiver	Original condition	Full merging
consumePureForall	✗	✓	✓	✓	✓
doubleAssignmentExists	✗	✓	✓	✓	✓
dutchFlag	✗	✓	✓	✓	✓
Listing 4.7	✗	✓	✓	✓	✓
Listing 4.8	✗	✗	✓	✓	✓
Listing 4.9	✗	✗	✗	✓	✓
parallelQsort	✗	✗	✓	✓	✓
seqSingleElements	✗	✗	✗	✓	✓
nodeVisited (0384a)	✗	✓	✓	✓	✓
qpWands	✗	✗	✗	✗	✗

Table 4.1: The table presents various examples that cause heap fragmentation. Each example is verified using each merging strategy individually, with the result recorded as either success (✓) or failure (✗).

Note that Table 4.1 also includes the three listings introduced in the explanation of the merging strategies in Section 4.4. Additionally, it contains `qpWands`, which could not be verified successfully, even with the full merging strategy. This demonstrates that while full merging is theoretically as complete as the over-approximating algorithm, there are still cases that cannot be verified in practice. Unfortunately, we were unable to determine the exact reason for this failure.

Nevertheless, Table 4.1 shows that the various strategies are highly effective, especially the original condition strategy was able to resolve most of the examples.

The next 5 out of the 18 cases clearly require reasoning across multiple

chunks due to disjunctions. These cases can only be resolved using the greedy approach with full merging enabled.

The last 3 cases are special corner cases and cannot be clearly classified under any specific cause. They are particularly challenging to resolve, and their underlying reasons are difficult to identify. Nevertheless, all of these cases could still be verified with full merging enabled.

This shows that while merging resolves many incompleteness issues, it cannot address all of them. The full merging strategy is a viable option in such cases, but it must be used with care, as demonstrated in Figure 4.11.

VerfiedSCION

To evaluate the practical applicability of the greedy approach and its merging strategies, we tested it on the `run` method in `dataplane.go`, which is one of the most ISC-heavy methods in VerifiedSCION, with the greedy approach combined with its merging strategies. Without any state consolidation the greedy algorithm was not able to verify the method successfully. Additionally, at one point, a disjunction involving permissions always requires falling back to the over-approximating algorithm. However, verification already failed before reaching that point when no state consolidation was applied.

While using only the tag system and/or the single-receiver strategy was not sufficient, the original condition proved to be the most effective. That said, we had to use a timeout of at least 100 milliseconds for its merging checks.

Overall, applying the greedy algorithm to `run` did not result in a decrease in total verification time, indicating that ISCs are not the main bottleneck in the verification process. However, a more detailed analysis revealed that the greedy algorithm did reduce the verification time related to handling ISCs, in one extreme case, by as much as eight seconds.

4.6.6 Summary

This concludes the evaluation of the greedy approach. Although the greedy read optimization did not perform as expected, we do believe that the greedy exhale algorithm is a viable alternative to the over-approximating one. The evaluation demonstrated reduced dependence on chunk processing order and significant performance improvements on selected examples. Furthermore, the proposed merging strategies enhance completeness while introducing only minimal overhead.

Chapter 5

General ISC improvements

This chapter explores various optimizations related to ISCs in Silicon. All improvements are independent of the ISC algorithm and can be applied to both the new greedy algorithm and the previous algorithm.

5.1 Optional summarization

The objective of this optimization is to minimize summarization while preserving completeness as much as possible.

5.1.1 Approach

Currently, Silicon summarizes the value maps of quantified chunks for any statement that involves the removal of permissions. The key insight behind this optimization is that summarization is only necessary when reasoning about values, not when dealing solely with permissions.

Each `exhale` statement in Listing 5.1 leads to a summarization of the quantified chunk values, adding extra quantified axioms to the path conditions. Details of how summarization is performed can be found in Algorithm 4.

The issue is that, by the final assertion, all summarization axioms are unnecessarily triggered. This leads to redundant instantiations proportional to the number of `exhale` statements times the number of chunks.

The key idea behind this optimization is to eliminate unnecessary instantiations for `exhale` and other statements where they are not needed. We prevent summarization when exhaling permissions for exhale statements or when exhaling method preconditions, asserting permissions, exhaling loop invariants, or when assigning a known value to a field. These operations do not directly depend on heap values, making summarization unnecessary in such cases.

```
field f: Int
function c(r: Ref) : Bool

method foo() {
    var x: Ref
    inhale c(x)
    inhale forall r: Ref ::
        c(r) ==> acc(r.f, write) && r.f == 0
    exhale forall r: Ref :: c(r) ==> acc(r.f, 1/5)
    // unnecessary summarization
    exhale forall r: Ref :: c(r) ==> acc(r.f, 1/5)
    // unnecessary summarization
    assert x.f == 0
}
```

Listing 5.1: Repeated summarization caused by multiple `exhale` statements.

While this optimization is effective, it introduces a potential incompleteness issue, as illustrated in Listing 5.2.

```
field f: Int
function c(r: Ref) : Bool

method foo() {
    var x: Ref
    inhale c(x)
    inhale forall r: Ref ::
        c(r) ==> acc(r.f, 1/2) && r.f == 0 // chunk 1
    // assert x.f == 0
    inhale forall r: Ref :: c(r) ==> acc(r.f, 1/3) // chunk 2
    exhale forall r: Ref :: c(r) ==> acc(r.f, 1/2)

    assert x.f == 0 // fails
}
```

Listing 5.2: Example where optional summarization leads to incompleteness.

This example produces two chunks with different permission amounts to the same locations. Additionally, only chunk 1 specifies a constraint on its values.

If, during `exhale`, chunk 1 is processed before chunk 2, it is entirely removed from the symbolic heap, causing the value constraint to be lost. As a result, the assertion fails due to missing value information whereas otherwise the summary would relate the values of both chunks when the exhale happens.

In practice, such an incompleteness issue is unlikely to occur. First, it requires a very specific chunk processing order, and second, other statements in the program also trigger summarizations preventing the lost of value constraints. Additionally, a user can manually prevent this with explicit assertions, as demonstrated by the commented-out assertion.

5.1.2 Implementation

In Silicon, this optimization is implemented by introducing an additional Boolean parameter to its internal exhale processing function. When this parameter is set to `true`, summarization is performed, when set to `false`, summarization is skipped.

By default, the flag is set to `false` for assignments, preconditions of method calls, assertions, and loop invariants, as these statements do not require summarization. Although a flag to disable this optimization has not been implemented, adding such an option in the future would be straightforward. The decision not to introduce a flag at this stage was made because incompleteness issues can always be resolved manually by adding explicit assertions.

5.1.3 Evaluation

This section demonstrates the effectiveness of the proposed optimization.

Experimental setup

The experiments were conducted on the Euler cluster at ETH Zurich [2], using a single CPU and 8 GB of RAM. Each benchmark program was verified 30 times with the Silicon backend, and the results were averaged across these runs. This evaluation includes programs of increasing complexity, as introduced in Section 4.6.1, along with typical ISC use cases from the Viper tutorial [5]. This section concludes with a practical completeness experiment, tested on smaller packages of VerifiedSCION.

Increasing complexity

Since this optimization primarily targets the reduction of quantified instantiations, we analyze how the number of instantiations decreases as program complexity increases. The programs used for this evaluation are the same as those introduced in Section 4.6.1.

The results in Figure 5.1 demonstrate the effectiveness of the optimization in reducing quantifier instantiations across different scenarios.

- `arraySum`: The optimization reduces quantifier instantiations by approximately 20%. Although this example involves numerous read

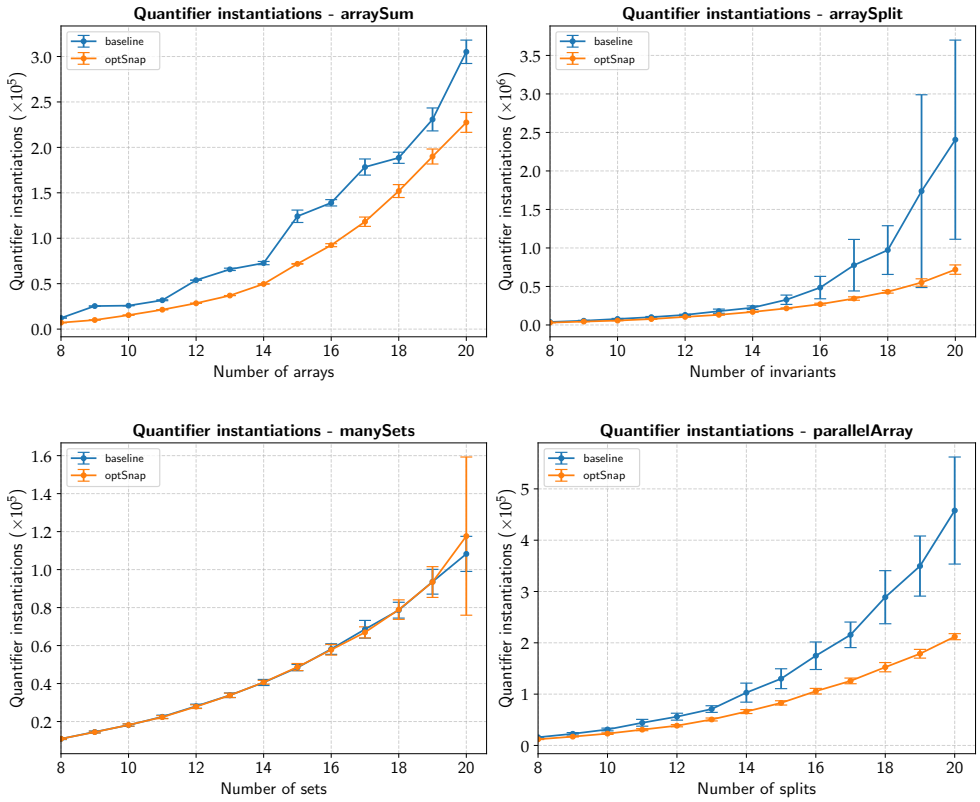

Figure 5.1: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

operations where the cache mechanism already minimizes summarization, the optimization still yields a noticeable improvement.

- `arraySplit`: A significant reduction in quantifier instantiations is observed. Due to the presence of many invariants, frequent summarizations occur, making this case particularly beneficial for the optimization. Unlike `arraySum`, caching is less effective here because the heap undergoes frequent modifications. Additionally, the results show greater stability, as indicated by the lower standard deviation.

- `manySets`: This example inherently generates very few summarizations, as it primarily consists of `inhale` statements. However, an interesting anomaly occurred with 20 sets: in one verification run, the process took 10 times longer than usual and instantiated 10 times more quantifiers, significantly impacting the standard deviation. Since more than 90% of the total verification time was spent on a single Z3 check, the most

likely explanation is an issue within Z3's internal heuristics.

- `parallelArray`: Quantifier instantiations generally decrease with the optimization, leading to more stable verification. This outcome was expected, as the example contains many `exhale` statements.

Overall, the results indicate that in examples that mostly exhale permissions, the optimization performs well in reducing quantifier instantiations.

Next, we evaluate whether the reduction of quantifier instantiations has a positive effect on overall verification times.

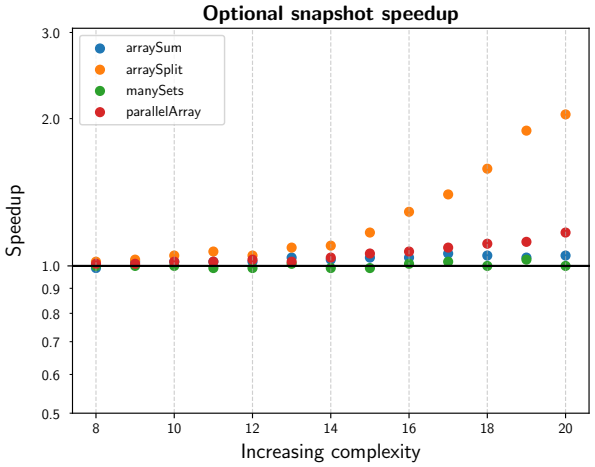

Figure 5.2: Speedup plot of examples. The X-axis represents the complexity level, while the Y-axis indicates the speedup. Data points represent the median verification time. The black line serves as the baseline reference at speedup = 1.

Note that we used the median, not the mean, to prevent outliers from skewing the results. As observed, most cases show a minor speedup, with an exceptional speedup of around 2 for `arraySplit`.

Overall, this demonstrates that the optimization reduces quantifier instantiations. If these are the primary source of slowdown, it can lead to a notable performance improvement as well.

General examples

Next, we evaluate the performance of this optimization against typical ISC use cases. Since these cases do not exhibit performance issues, a speedup is not expected. However, a reduction in quantifier instantiations is still possible.

Most examples in Table 5.1 are verified with a reduced number of quantifier instantiations and the optimization is particularly effective for `arrayList`,

Examples	Baseline (Mean)	Optimized (Mean)	Baseline (CoV)	Optimized (CoV)
arrayList	3.06×10^4	2.54×10^4	44.73	26.59
arrayMaxElimination	2.19×10^3	2.00×10^3	1.23	0.80
dutchFlag	1.93×10^3	1.78×10^3	2.73	0.83
binarySearch	1.49×10^3	1.36×10^3	1.08	1.03
graphMarking	2.05×10^4	2.05×10^4	0.00	0.00
graphCopy	3.25×10^4	3.38×10^4	1.85	5.21
arrayMaxStraightForward	1.10×10^3	812.40	1.53	0.51
longestCommonPrefix	669.04	570.14	2.18	0.49
parallelArray	816.00	834.00	0.00	0.00

Table 5.1: Mean and coefficient of variation (CoV) of quantifier instantiations for various examples, both with and without optimization. We used CoV instead of standard deviation, as interpreting standard deviation becomes challenging with such large values.

where it significantly decreases quantifier instantiations and improves the CoV. However, there is also an example, `graphCopy`, where the optimization unexpectedly increases quantifier instantiations and introduces instability.

Completeness

While it is difficult to make definitive statements about completeness, this optimization was applied to verify VerifiedSCION, resulting in only a single new verification error, which was easily resolved by adding an additional assertion. From this, we can infer that the theoretical incompleteness of this optimization is unlikely to occur frequently in practice.

Preliminary tests on `scion` and `slayers` showed a reduction of approximately 5% to 10% in quantifier instantiations. However, these reductions did not lead to noticeable improvements in verification time suggesting that quantifier instantiations are not the primary bottleneck. Nonetheless, the decrease in instantiations is still a meaningful improvement.

5.2 Summarization triggers

The goal of this optimization is to reduce the number of triggered summarization axioms without compromising completeness. The key insight is that using more specific triggers can limit unnecessary instantiations without impacting completeness in most practical cases.

5.2.1 Approach

An interesting fact about SMT solvers like Z3 is that the order in which quantified axioms and triggering terms appear does not affect the logical

outcome. This principle is illustrated in Listing 5.3, which shows two SMT-LIB snippets. One uses forwards triggering, where the term that initiates a quantified formula appears before the formula itself, while the other uses backwards triggering, where the quantified formula is introduced before the triggering term. Both lead to the same output.

```
(declare-const x Int)
(declare-fun f (Int) Bool)
(assert (forall ((y Int))
    (!
     (=> (f y) (> y 0))
     :pattern (f y))))
(assert (f x)) ; Triggers
    the quantified formula
(assert (< x 0))
(check-sat)
Output: unsat
```

```
(declare-const x Int)
(declare-fun f (Int) Bool)
(assert (f x)) ; Triggers
    the quantified formula
(assert (forall ((y Int))
    (!
     (=> (f y) (> y 0))
     :pattern (f y))))
(assert (< x 0))
(check-sat)
Output: unsat
```

Listing 5.3: The left snippet demonstrates backwards triggering, where the quantified formula is emitted before the triggering term. The right snippet demonstrates forwards triggering, where the quantified formula is emitted after the term that triggers it. Despite the different order, both are logically equivalent.

From a purely logical perspective, formulas do not have a control flow. They are declarative, meaning their order should not affect their satisfiability. However, in imperative programming languages, the order of statements does matter. The key insight is that this before-after relationship can be leveraged to reduce instantiations by selecting trigger patterns that only match ground terms which were introduced after the quantified axiom had been emitted.

Let us consider the quantified axiom and its trigger that Silicon adds to the path conditions during a summarization for chunk $\mathtt{ch}_i$. Details about this axiom can be found in Section 2.2.3 and Algorithm 4. For clarity, we briefly reiterate the meaning of the variables, $ch_i.v_i$ refers to the chunk's value map, $ch_i.p_i$ to its permission function, and v denotes a fresh value map.

$$\forall r \in Ref.\ \{ch_i.v_i(r)\}\ \{v(r)\}\ 0 < ch_i.p_i(e^{-1}(r)) \implies v(r) = ch_i.v_i(r) \quad (5.1)$$

Equation 5.1 contains two triggers. The first trigger, $\{ch_i.v_i(r)\}$, is a more precise trigger because it is unique for each summarization. Consequently, it can only occur in ground terms introduced after the axiom's definition, ensuring that only backwards triggering occurs. The second trigger, $\{v(r)\}$, can also be triggered through forwards triggering since all summarization

axioms related to the chunk share this exact same trigger. This means that if one axiom is instantiated by a ground term, it will cascade and trigger all other axioms as well.

Relying solely on the first trigger results in a more controlled and explicit instantiation of axioms, significantly reducing unnecessary instantiations.

Let us illustrate this scenario with a Viper example:

```
1
2  field f: Int
3  function c(r: Ref) : Bool
4
5  method foo() {
6      inhale forall r: Ref :: c(r) ==> acc(r.f, write)
7      val := x.f
8      exhale forall r: Ref :: c(r) ==> acc(r.f, 1/10)
9      exhale forall r: Ref :: c(r) ==> acc(r.f, 1/10)
10     exhale forall r: Ref :: c(r) ==> acc(r.f, 1/10)
11     exhale forall r: Ref :: c(r) ==> acc(r.f, 1/10)
12 }
```

Listing 5.4: Repeated summarizations are triggered by the assignment `val := x.f` due to forwards triggering, leading to unnecessary instantiations.

To better understand what is happening in Listing 5.4, let us name the chunk ch_1 that is created by the `inhale` statement. The assignment is then defined as $val = ch_1.v_1(x)$. After each exhale statement, summarization is performed, introducing a fresh snapshot map, which we denote as v_k along with an axiom as defined in Equation 5.1.

If both triggers are used, we observe that the assignment is syntactically equivalent to the second trigger $ch_i.v_i(r)$, causing all summarization axioms to be triggered. This results in forwards triggering, potentially leading to unnecessary overhead.

With only the first trigger, on the other hand, only the summarizations introduced before the assignment are triggered, since all $v_k(r)$ are not syntactically equal to $ch_1.v_1(r)$. This helps avoid extra overhead. However, relying exclusively on backwards triggers can cause incompleteness in specific cases.

Since the assertion in Listing 5.5 is quantified, it does not trigger the summarization axioms directly, even though they are required to prove the assertion. This summarization is necessary to connect the chunk created by the first `inhale` with the value constraint introduced by the second `inhale`. If only backwards triggering is used, the solver never instantiates this summarization, resulting in a failed proof. In contrast, forwards triggering ensures

```
field f: Int

method foo() {
    var x: Ref
    inhale forall r: Ref :: x == r ==> acc(r.f, write)
    inhale x.f == 5
    // summarization
    assert !(forall r: Ref :: x == r ==> r.f == 5)
}
```

Listing 5.5: Example where using only backwards triggering results in reduced completeness.

that the summarization axioms are instantiated, enabling the SMT solver to successfully verify the assertion.

In practice, the completeness of backwards triggering depends on the specific program. While missing instantiations can be manually triggered by adding additional assertions, it may not always be intuitive why certain programs cannot be verified. Therefore, this optimization should be configurable, allowing users to enable or disable it as needed.

5.2.2 Implementation

The implementation of this approach is straightforward. The forwards trigger $v_i(r)$ is removed from the summarization axiom definitions, ensuring that only backwards triggering occurs.

Additionally, the flag `--SummarizationWithoutForwardsTrigger` is provided, allowing users to enable or disable this optimization as needed.

5.2.3 Evaluation

This section assesses the effectiveness of the proposed optimization, using the same experimental setup as described in Section 5.1.3.

Increasing complexity

We begin by evaluating the behavior of quantifier instantiations as program complexity increases. Since this optimization, while working in a different way, targets the same aspect as optional summarization, we will additionally compare the results not only against the baseline but also against optional summarization.

As shown in Figure 5.3, the optimization outperforms the baseline in three out of the four examples. Interestingly, its performance is nearly identical to that of optimal summarization. A detailed analysis of the concrete instantiations revealed that both optimizations prevent the exact same instantiations.

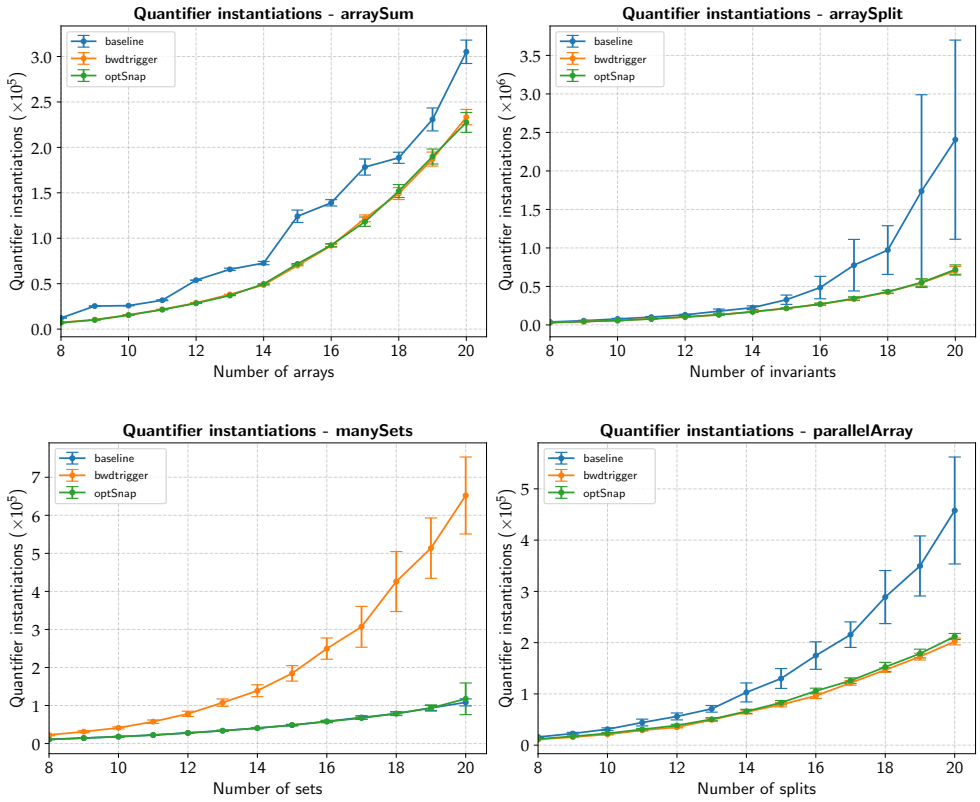

Figure 5.3: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

However, optimal summarization avoids the creation of axioms altogether, whereas this optimization still adds the axioms but prevents them from being unnecessarily triggered. Since both optimizations yield similar verification times, we have not included an additional graph for verification times.

Overall, this demonstrates that both techniques produce similar results. This allows for flexibility in choosing between the optimizations depending on the use case, for example, if one encounters an incompleteness issue. However, applying both simultaneously does not provide any additional benefits.

Now, let us analyze the results for `manySets`. Unexpectedly, the number of quantifier instantiations increased significantly. However, an analysis of the data did not reveal any clear reason for this behavior.

Interestingly, as shown in Figure 5.4, the impact on verification time is minimal, but overall Z3's memory usage is reduced by a third. Although the exact reason for this behavior is unclear, one possible explanation might be

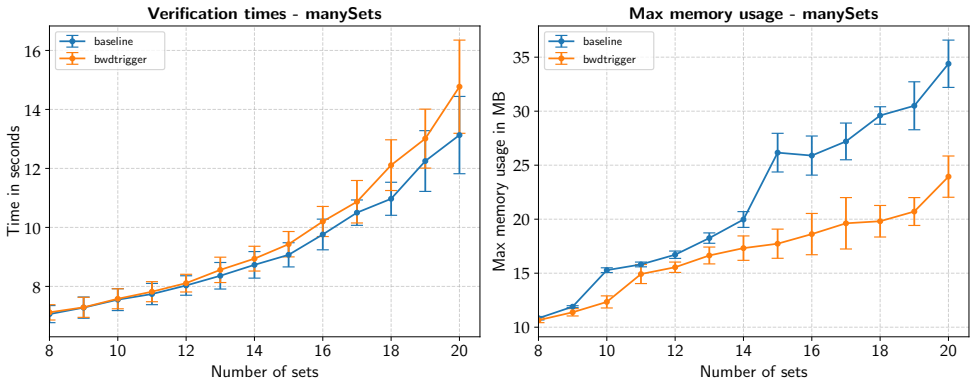

Figure 5.4: Verification times (left) and memory usage in megabytes (MB) (right) for `manySets` compared to the baseline with an increasing number of sets. The X-axis represents the number of sets, while the Y-axis shows verification time in seconds (left) and memory usage (right). Data points are averaged, with standard deviation included to indicate stability.

that limiting quantified axioms to a single trigger causes Z3 to switch to a different heuristic, possibly one that instantiates more aggressively. However, this remains speculative and would require further investigation to confirm.

General examples

Since the previous experiment revealed that this optimization reduces memory usage, we now evaluate whether this effect consistently holds for general ISC examples.

Examples	Baseline (Mean)	Optimized (Mean)	Baseline (Std)	Optimized (Std)
arrayList	47.29	17.97	60.50	32.80
arrayMaxElimination	4.62	4.47	0.13	0.02
dutchFlag	4.89	4.88	0.06	0.06
binarySearch	4.34	4.33	0.01	0.01
graphMarking	9.49	9.21	0.00	0.00
graphCopy	9.54	9.24	0.05	0.05
arrayMaxStraightForward	4.27	4.03	0.00	0.00
longestCommonPrefix	4.07	4.05	0.01	0.00
parallelArray	4.32	4.32	0.00	0.00

Table 5.2: Mean and standard deviation (Std) of memory usage in megabytes (MB) for various examples, both with and without optimization.

As shown in Table 5.2, memory usage remains low in most cases. However, in the example with high memory consumption, the optimization significantly reduces memory usage, confirming its positive impact on Z3's max memory usage.

Completeness

To provide an overview of how frequently incompleteness occurs in practice, the optimization was tested again with VerifiedSCION. Compared to optional summarization, this optimization experiences more instances of incompleteness. The verification of both `scion` and `slayers` resulted in two verification errors each. Preliminary investigations did not reveal an immediate solution, and no further time was spent on resolving these issues. Additionally, verifying the `dataplane.go` file resulted in eight more verification errors, which were not further investigated. These results suggest that this optimization may lead to incompleteness more quickly.

However, it may still be worthwhile to analyze and resolve these verification errors in the future as quantifier instantiations for `slayers` and verification time for `dataplane.go` were noticeably reduced. That said, these numbers must be interpreted with caution. Since verification errors significantly alter the verification process, it remains unclear how the results would look without any errors. Unfortunately, without first resolving the verification errors, there is no way to make a definitive assessment.

5.3 Alternative axiom definitions

This optimization aims to identify and remove redundant terms in the definitions of permission reasoning, thereby simplifying proof obligations for the SMT solver. By redefining certain axioms, the approach seeks to minimize complexity while still preserving completeness as much as possible.

5.3.1 Approach

Let us begin by restating the definitional axioms for the inverse functions and the resulting chunk representation. Detailed information on these definitions can be found in Section 2.2.2.

$$\forall x : \mathcal{T}.\ c(x) \implies img(e(x)) \wedge e^{-1}(e(x)) = x \tag{5.2}$$

$$\forall r : Ref.\ c(e^{-1}(r)) \wedge img(r) \implies e(e^{-1}(r)) = r \tag{5.3}$$

$$[r.f \rightarrow v(r)\ \#\ img(r) \wedge c(e^{-1}(r))\ ?\ p(e^{-1}(r)) : 0] \tag{5.4}$$

A recurring term in all three definitions is the condition function $c(x)$. The goal of this optimization is to eliminate this redundancy.

Under the assumption that:

$$\forall x : \mathcal{T}.img(e(x)) \implies c(x) \tag{5.5}$$

we can simplify the representation by replacing every occurrence of $img(r) \wedge c(e^{-1}(r))$ with just $img(r)$. This removes the redundant condition function while preserving correctness.

However, with the current inverse function axioms, this assumption does not necessarily hold. The image function $img(r)$ is not strictly defined by the axioms as a traditional image function in a mathematical sense. Consider the following interpretation:

$$\forall r \in Ref. \quad img(r) = true \tag{5.6}$$

While this interpretation is nonsensical from a practical standpoint, it is technically valid under the current definitions. However, it clearly violates our assumption in Equation 5.5, proving that the assumption does not hold.

Even though our assumption is false, it is still sound to remove the condition function. This is because the opposite implication:

$$\forall x : \mathcal{T}.img(e(x)) \impliedby c(x) \tag{5.7}$$

does hold due to Equation 5.2. However, this modification introduces incompleteness in reasoning about permissions. The proof obligation for verifying whether a location lacks permission changes from $\neg img(r) \vee \neg c(r)$ to $\neg img(r)$ which is harder to prove, if not impossible in most cases.

```
1
2 field f: Int
3 function c(r: Ref) : Bool
4
5 method foo() {
6     inhale forall r : Ref :: c(r) ==> acc(r.f, write)
7     assert forall r : Ref :: !(c(r)) ==> perm(r.f) == none
8 }
```

Listing 5.6: Example where removing the condition function leads to incompleteness.

Listing 5.6 illustrates how omitting the condition function can lead to incompleteness. In this scenario, the assertion hinges on proving $\neg c(r) \implies \neg img(r)$, a fact the SMT solver is unable to establish on its own. However, when permissions are also restricted by the condition function, the resulting proof obligation becomes $\neg c(r) \implies \neg img(r) \vee \neg c(r)$, which is trivially true.

The issue arises because the definitional axioms do not specify under what conditions the image function should be *false*. To address this, we can strengthen the definitions so that $img(r)$ behaves more like a traditional image function would do in mathematics.

$$\begin{aligned}\forall x : \mathcal{T}.\{\ img(e(x))\ \}\ (c(x) \implies img(e(x)) \wedge e^{-1}(e(x)) = x) \wedge \\ (\neg c(x) \implies \neg img(e(x)))\end{aligned} \tag{5.8}$$

The alternative definition in Equation 5.8 aligns with the mathematical notion of an image function. As a result, we can prove the assumption in Equation 5.5, allowing us to remove $c(x)$ from (5.3) and (5.4) without compromising completeness. While this holds in theory, in practice, the removal of $c(x)$ may lead to missing instantiations, as a relevant ground term is no longer available to trigger necessary axioms. As shown in Equation 5.8 the trigger must now be $img(e(x))$, whereas previously $c(x)$ could also serve as a trigger.

Listing 5.7 provides a concrete illustration where the alternative axiomatization is less complete.

```
1  predicate p(i:Int)
2
3  method fail(n: Int)
4      requires forall i: Int ::
5          i in [1..n+1) ==> acc(p(i), write)
6      ensures  forall i: Int ::
7          i in [0..n) ==> acc(p(i+1), write)
8  {}
9
10 function add(i: Int, j: Int) : Int
11 { i + j }
12
13 method success(n: Int)
14     requires forall i: Int :: i in [1..n+1) ==> acc(p(i), write)
15     ensures  forall i: Int ::
16         i in [0..n) ==> acc(p(add(i,1)), write)
17 {}
```

Listing 5.7: Example where the alternative axiomatization is less complete, depending on whether built-in or custom arithmetic functions are used.

Note that both methods in Listing 5.7 have empty bodies, {}, meaning they are not abstract and must be fully verified. The `fail()` method cannot be verified under the alternative axiomatization because the quantified variable `i` uses the built-in arithmetic operation +. This complicates trigger inference in Silicon, as arithmetic operations are not allowed in triggers. Under the

original axiomatization, the condition itself, `i in [0..n)`, can serve as a trigger, which is sufficient for successful verification. However, in the alternative approach, this trigger is never triggered, causing the incompleteness. Unfortunately, there is no trigger a user can provide that resolves this issue.

In contrast, `success()` uses a custom `add()` function, which allows Silicon to infer a more suitable trigger, in this case, `img(p(add(i,1)))`, enabling successful verification. In summary, the proposed change can lead to triggering-related incompletenesses if the image function cannot be used as a trigger, whereas the old encoding can, in those cases, rely on triggering terms from the condition function.

This demonstrates that the alternative axiomatization, while reducing redundant terms, can, in some cases, lead to incompleteness due to trigger complexities, particularly when built-in arithmetic is used. Deciding which axiomatization to use is, once again, a trade-off between performance and completeness.

5.3.2 Implementation

The implementation of this optimization involves minor changes in several areas. First, the definitions of inverse functions must be updated according to the approach described above. Second, the chunk's definition no longer includes the condition function. Finally, whenever quantified permissions are exhaled, the permission removal function omits the condition as well.

Because incompleteness issues can arise in some cases,
a `--AltInvAxiomatization` flag is provided, giving users the option to enable or disable this feature as needed.

5.3.3 Evaluation

This section evaluates the proposed optimization. In addition to quantifier instantiations, we also analyze conflicts, a metric that reflects the complexity of solving constraints in Z3. Since this optimization aims to simplify constraints, the goal is to observe a reduction in this metric as well. The experimental setup is identical to that described in Section 5.1.3.

Increasing complexity

We start by assessing the reduction in quantifier instantiations for the familiar set of programs with increasing levels of complexity.

Figure 5.5 shows a significant reduction in quantifier instantiations, with particularly strong improvements for `manySets` and `parallelArray`. A closer analysis reveals that complex permission expressions are now simplified due to the removal of conditions, especially in programs involving many inhales

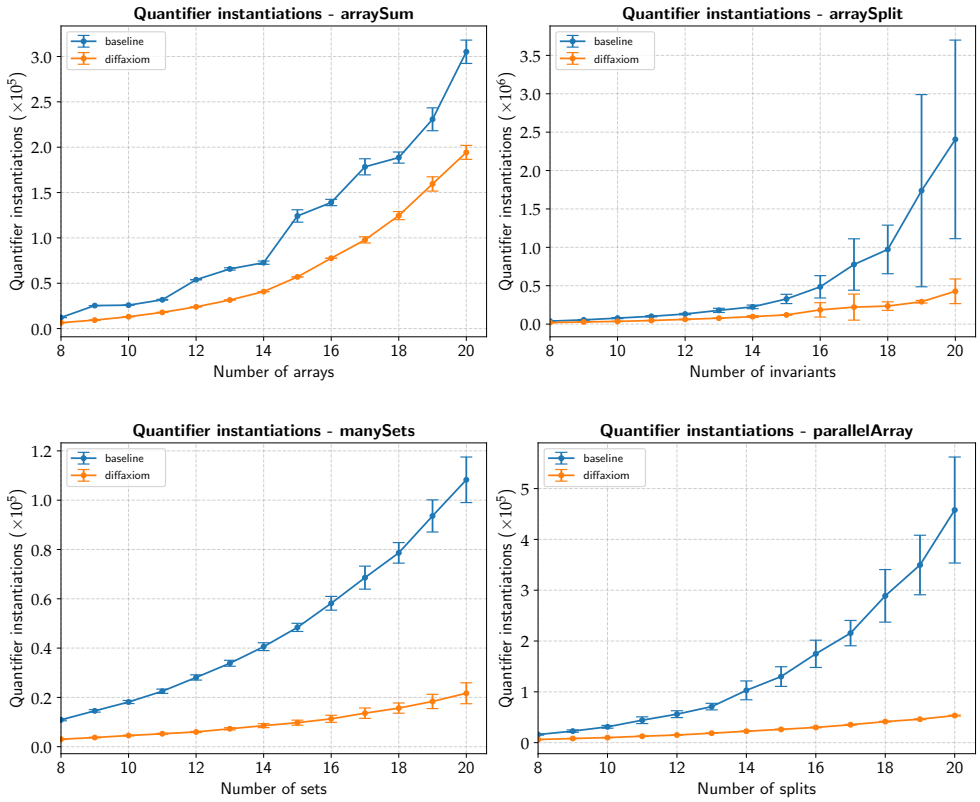

Figure 5.5: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

and exhales. This optimization proves highly effective, both in reducing the overall number of instantiations and in improving stability, as indicated by a lower standard deviation.

We can conform this further by analyzing the number of conflicts.

As shown in Figure 5.6, the number of conflicts is reduced in three of the four examples. Interestingly, for `arraySum`, the conflict count remains nearly constant and even slightly increases at 20 arrays. This is likely because the ISCs in `arraySum` have simple conditions, and the permission expressions of chunks remain simple, as the code in the example always exhales permission amounts corresponding to entire heap chunks instead of removing partial permission amounts.

As a final evaluation step, we examine the achieved speedups. The results in Figure 5.7 show speedups across all examples. Once again, we used the median rather than the mean to calculate speedup. An exceptional speedup

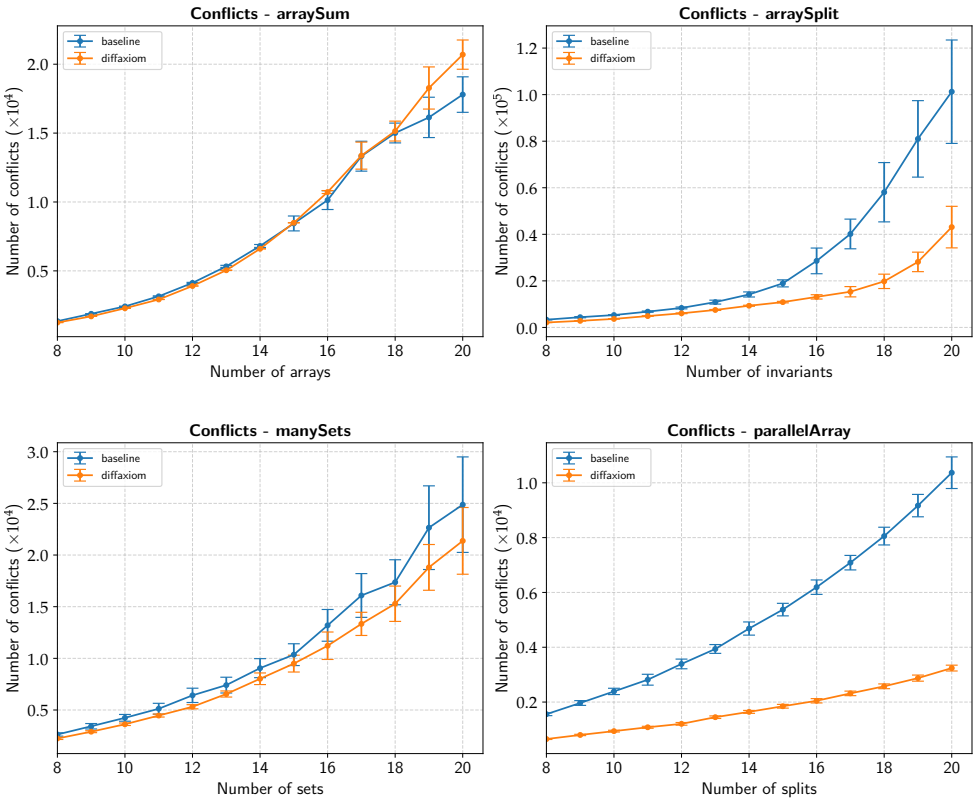

Figure 5.6: Number of conflicts for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of conflicts. Data points are averaged, and the standard deviation is included to indicate stability.

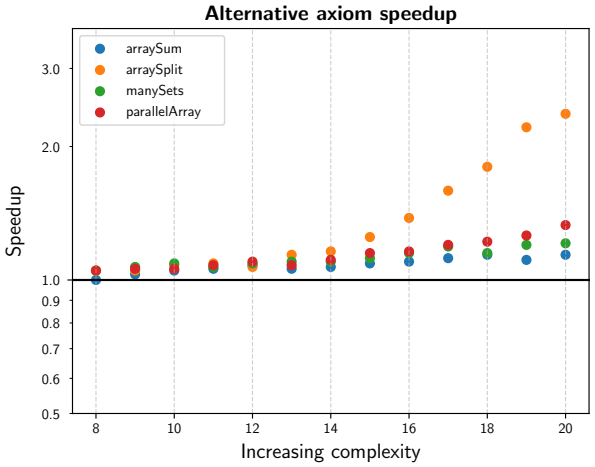

Figure 5.7: Speedup plot of examples. The X-axis represents the complexity level, while the Y-axis indicates the speedup. Data points represent the median verification time. The black line serves as the baseline reference at speedup = 1.

of more than 2 is observed for `arraySplit`, demonstrating the effectiveness of this optimization for certain programs.

General examples

Next, we evaluate how the quantifier instantiations metric performs in typical programs that use ISCs.

Examples	Baseline (Mean)	Optimized (Mean)	Baseline (CoV)	Optimized (CoV)
arrayList	3.06×10^4	3.66×10^4	44.73	48.38
arrayMaxElimination	2.19×10^3	1.61×10^3	1.23	2.72
dutchFlag	1.93×10^3	1.93×10^3	2.73	7.32
binarySearch	1.49×10^3	1.19×10^3	1.08	0.79
graphMarking	2.05×10^4	1.44×10^4	0.00	0.26
graphCopy	3.25×10^4	3.49×10^4	1.85	8.51
arrayMaxStraightForward	1.10×10^3	743.42	1.53	0.07
longestCommonPrefix	669.04	425.72	2.18	2.45
parallelArray	816.00	696.00	0.00	0.00

Table 5.3: Mean and coefficient of variation (CoV) of quantifier instantiations for various examples, both with and without optimization.

The notable result from Table 5.3 is that quantifier instantiations increase in two while decreasing in six. A closer analysis indicates that the optimization is particularly sensitive to trigger selection. With appropriate triggers, it can perform very well. However, suboptimal triggers may lead to increased instantiations. Since these examples have not been altered, it is likely that choosing different triggers would yield better results, but this possibility has not been further explored.

Completeness

Applying this optimization to VerifiedSCION produced suboptimal results. In the `scion` package, five new verification errors were introduced, and verification time was more than doubled. However, this should be interpreted with caution, as verification errors can significantly increase verification time. Quantifier instantiations remained roughly the same. The `slayers` package was successfully verified but did not show any significant improvements in either verification time or quantifier instantiations. Unfortunately, verification of `dataplane.go` resulted in a timeout.

Overall, these results suggest that the optimization is more sensitive to trigger selection than the original approach, which is likely why it did not perform well on VerifiedSCION. However, when applied with well-chosen triggers, it can significantly improve performance, as demonstrated in other examples.

5.4 Memoization of inverse functions

This section presents an optimization that memoizes inverse function definitions to prevent the creation of multiple identical inverse functions for similar ISCs. Instead of defining a new inverse function each time, memoized function definitions can be reused.

5.4.1 Approach

Typically, similar ISCs are used in different parts of a program. As shown in Listing 5.8, an ISC may be provided to a method via a precondition, then used in a loop invariant, and finally returned via a postcondition. As a result, Silicon defines multiple inverse functions that are semantically identical. The goal of this optimization is to detect these cases and reuse previously created definitions. Listing 5.8 illustrates such recurring patterns of similar ISCs.

```
1  field f: Int
2  method foo(s: Set[Ref])
3      requires forall r: Ref :: r in s ==> acc(r.f, write)
4      ensures forall r: Ref :: r in s ==> acc(r.f, write)
5  {
6      while( /* ... */ )
7          invariant forall r: Ref ::
8              r in s ==> acc(r.f, write)
9      { /* ... */ }
10 }
```

Listing 5.8: Repeated ISC usage in different parts of a program. The same ISC appears in a precondition, loop invariant, and postcondition, leading to redundant inverse function definitions.

In order to detect such situations, we first need to determine whether two ISCs can share the same inverse function definitions. To do so, we must identify the constraints that must hold between two ISCs for their inverse function axioms to be identical.

Consider the following two ISCs. We aim to determine the constraints under which the definitional axioms of the first ISC (5.9) can be reused for the second one (5.10).

$$\forall i \in \mathcal{T}.\, c_1(i) \implies acc(e_1(i).f, p_1(i)) \tag{5.9}$$
$$\forall i \in \mathcal{T}.\, c_2(i) \implies acc(e_2(i).f, p_2(i)) \tag{5.10}$$

Their inverse function definitions are equal if and only if $c_1(i) = c_2(i)$ and $c_1(t) \implies e_1(t) = e_2(t)$. In other words, the condition functions and

the receiver functions must be equal. However, it is independent of the permission function. Attempting to weaken these requirements to something like $c_1(i) \implies c_2(i)$ and $e_1(\mathcal{T}) \supseteq e_2(\mathcal{T})$ is unsound, as illustrated by the following example:

```
1 field f: Int
2 function e(i: Int) : Ref // assumed to be injective
3 method foo() {
4     inhale forall i : Int :: true ==> acc(e(i).f, write)
5     exhale forall i : Int :: true ==> acc(e(2*i).f, write)
6     // succeeds with weaker requirements
7     assert perm(e(1).f) == none
8 }
```

Listing 5.9: Unsound example if weaker requirements were used to reuse inverse functions.

In Listing 5.9, both ISCs trivially satisfy $\mathit{true} \implies \mathit{true}$ and $e_1(\mathcal{T}) \supseteq e_2(\mathcal{T})$ but not $e_1(t) = e_2(t)$, meaning we only reuse inverse function definition axioms if we use the weaker requirements.

The issue arises from the fact that the receiver expressions are entirely different, and therefore their inverse functions are also different. To make this more concrete, if we replace the second ISC's axioms with those of the first, it implies that both ISCs share the same image function. Since the condition function is always *true*, the permissions available to locations are restricted solely by the image function. This, in turn, implies that both ISCs represent identical permissions to locations, which is clearly incorrect. As a result, this leads to unsoundness, as demonstrated by the final assertion.

This demonstrates the requirement of the stronger equality constraints for ISCs. While reusing inverse function definitions prevents the introduction of new ones, it may increase the frequency of reused axioms being triggered. However, this comes with a significant advantage. Without memoization, the SMT solver must explicitly prove the equality between inverse functions, whereas memoization ensures they are syntactically identical. This reduction simplifies the solver's proof obligation and reduces overall instantiations, ultimately improving verification performance.

5.4.2 Implementation

The implementation of this optimization extends Silicon's state with a map that stores previously defined inverse functions. This map uses a tuple consisting of a condition function and a receiver function as the key, and returns the corresponding inverse function. Whenever a new inverse function needs to be defined, Silicon checks whether an entry with the key $(c(x), e(x))$

already exists in the map. If a matching entry is found, it reuses the existing functions. Otherwise, it defines new axioms and adds a new entry with the inverse functions to the map.

There are two key details worth noting. First, the map lookup is purely syntactic. While, in theory, it is possible to use aliasing information from the SMT solver for condition and receiver function arguments, this would result in many SMT solver checks, negating all performance gains.

The second challenge is that Silicon does not currently enforce a consistent naming scheme for quantified variables. Although the specific names of quantified variables are semantically irrelevant, they impact syntactic equality checks. To address this, we modified Silicon to rename all quantified variables according to a fixed naming scheme, making lookup checks possible.

As a side note, while this renaming is currently implemented only for lookups in the inverse function map, adopting a fixed naming scheme for quantified variables globally would be beneficial. It would help prevent redundant constraints arising from structurally identical terms that differ only in variable names, which can occur in cases such as chunk properties.

5.4.3 Evaluation

This section evaluates the effectiveness of the previously introduced feature. The experimental setup is the same as in Section 5.1.3.

Increasing complexity

We begin by analyzing the impact of memoization on quantifier instantiations as program complexity increases.

The results in Figure 5.8 show mixed outcomes. While memoization significantly reduces quantifier instantiations for `parallelArray`, it increases instantiations and reduces stability for `arraySplit`, as indicated by the large standard deviation. Since `manySets` consists only of ISCs with unique conditions, caching is not applied, leading to no effect. Additionally, the improvement for `arraySum` is smaller than expected, despite it being the primary example with many similar ISCs.

Overall, the results are not as expected, showing greater instability than anticipated. Although pinpointing the exact cause of this instability is challenging, it is likely that certain internal Z3 heuristics perform more effectively when generating new axioms rather than reusing previously created ones.

General examples

We aim to further investigate this instability and determine whether there is a causal relationship between a high number of cached functions and increased

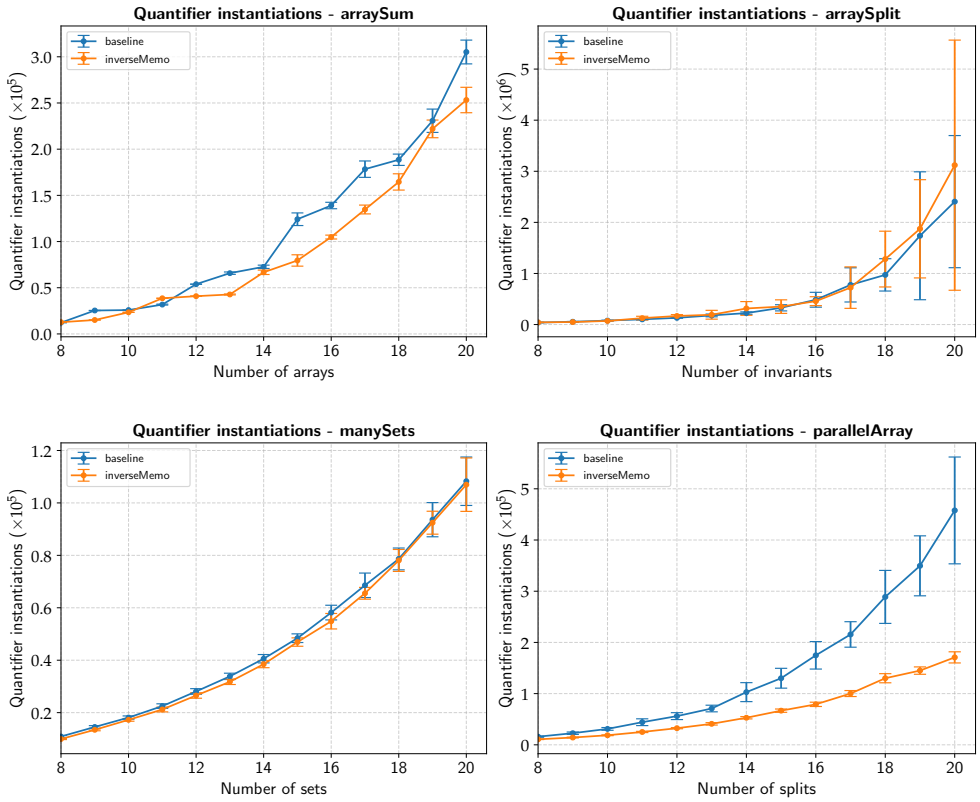

Figure 5.8: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

instability in quantifier instantiations. Additionally, we assess how often caching occurs in practice.

Examples	# of cached inverse functions	Baseline (Mean)	Optimized (Mean)	Baseline (CoV)	Optimized (CoV)
arrayList	6 / 18	3.06×10^4	3.03×10^4	44.73	59.40
arrayMaxElimination	9 / 11	2.19×10^3	1.73×10^3	1.23	2.21
dutchFlag	7 / 8	1.93×10^3	1.69×10^3	2.73	5.46
binarySearch	7 / 8	1.49×10^3	1.25×10^3	1.08	0.87
graphMarking	45 / 48	2.05×10^4	1.19×10^4	0.00	0.00
graphCopy	11 / 36	3.25×10^4	3.02×10^4	1.85	0.49
arrayMaxStraightForward	9 / 11	1.10×10^3	841.70	1.53	0.32
longestCommonPrefix	5 / 6	669.04	443.00	2.18	6.90
parallelArray	8 / 14	816.00	709.14	0.00	0.14

Table 5.4: Number of cached inverse functions relative to the total number of inverse functions. Additionally, the table includes the average number of quantifier instantiations and their coefficient of variation (CoV).

From these examples, it is evident that memoization is frequently applied in typical ISC scenarios. However, while cases with a high number of cached inverse functions, such as `graphMarking`, show a nearly 50% reduction in quantifier instantiations, this usually comes at the cost of increased instability. In almost all cases, caching leads to greater variability, with `arrayList` being an extreme example, exhibiting a significantly higher CoV.

Completeness

The observed instability in smaller examples raises concerns about applying this optimization to VerifiedSCION. Both verification attempts for the `scion` and `slayers` packages resulted in timeouts, indicating that these instabilities become even more pronounced in larger examples. Given the issues revealed by this evaluation, we recommend using this optimization only for smaller programs and with caution.

5.5 Combined evaluation

As a final experiment, we aim to evaluate the combined effect of all optimizations, as their individual improvements do not necessarily accumulate. To do this, we use the same setup as in Section 5.1.3, but with all optimizations applied together. However, we chose not to run this experiment on VerifiedSCION, as the individual optimizations already exhibited incompleteness issues. A combined approach would likely face the same challenges, rendering the experiment uninformative.

Increasing complexity

We begin by analyzing how the combined optimizations affect quantifier instantiations as program complexity increases.

As expected, Figure 5.9 shows a significant reduction in quantifier instantiations across all programs. However, the instability issues related to summarization triggers in `manySets` remain noticeable. This suggests that if one optimization introduces issues, it can also impact others. Overall, the combined optimizations complement each other well, improving different aspects of the verification overhead.

We proceed by analyzing how these improvements impact speedup. Once again, we emphasize that the median is used to measure speedup in Figure 5.10. All examples experience a speedup, with a particularly significant improvement for `arraySplit`. At 20 splits, it achieves a speedup of nearly 3, making it a prime example of the effectiveness of combining all optimizations. While other examples do not show such a dramatic improvement in verifi-

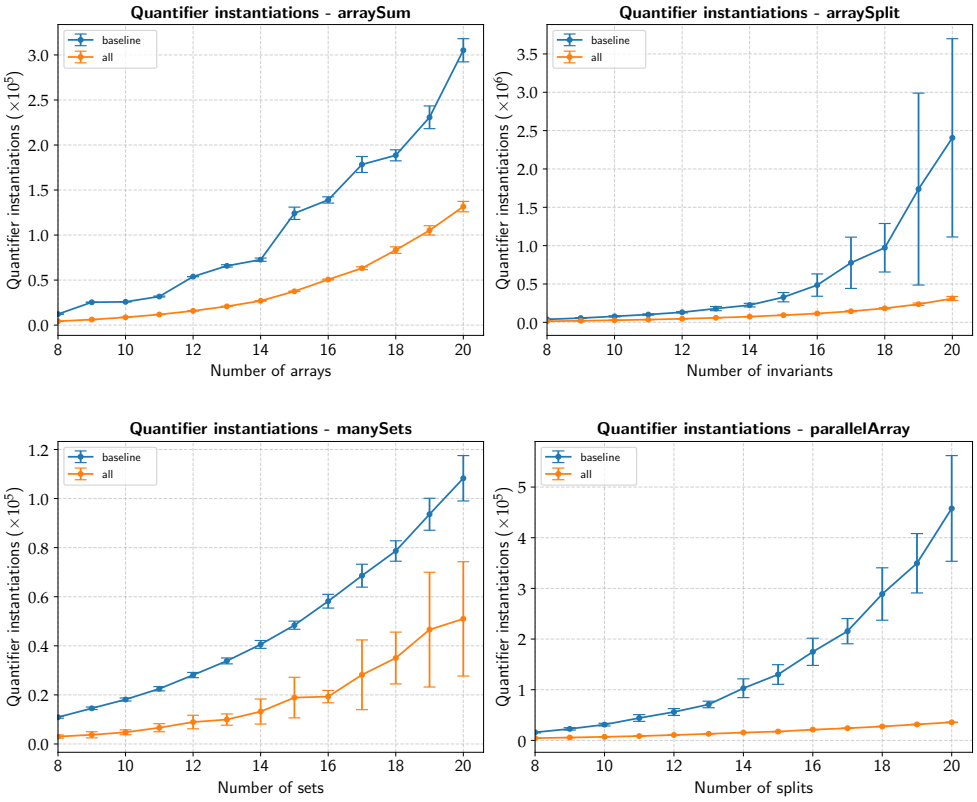

Figure 5.9: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

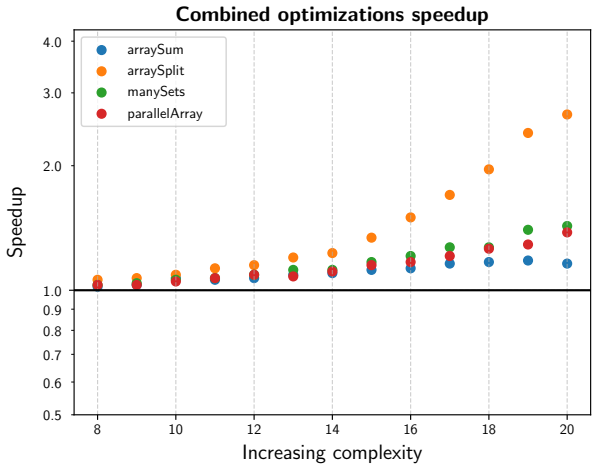

Figure 5.10: Speedup plot of examples. The X-axis represents the complexity level, while the Y-axis indicates the speedup. Data points represent the median verification time. The black line serves as the baseline reference at speedup = 1.

cation times, the overall trend clearly indicates that as program complexity increases, the speedup also grows.

General examples

Next, we assess how the number of quantifier instantiations is affected in typical ISC programs.

Examples	Baseline (Mean)	Optimized (Mean)	Baseline (CoV)	Optimized (CoV)
arrayList	3.06×10^4	1.91×10^4	44.73	62.68
arrayMaxElimination	2.19×10^3	1.04×10^3	1.23	1.14
dutchFlag	1.93×10^3	1.47×10^3	2.73	4.54
binarySearch	1.49×10^3	810.22	1.08	1.43
graphMarking	2.05×10^4	2.10×10^4	0.00	5.52
graphCopy	3.25×10^4	2.72×10^4	1.85	4.73
arrayMaxStraightForward	1.10×10^3	526.00	1.53	0.00
longestCommonPrefix	669.04	221.44	2.18	1.70
parallelArray	816.00	555.00	0.00	0.00

Table 5.5: Mean and coefficient of variation (CoV) of quantifier instantiations for various examples, both with and without optimization.

Table 5.5 shows that across all benchmarked examples the number of quantifier instantiations has been significantly reduced. The only exception is `graphMarking`, which exhibits a slight increase, though the absolute numbers remain negligible.

This concludes our evaluation of the combined optimizations. The primary goal, reducing quantifier instantiations, was largely met, and in most cases, this reduction led to a modest speedup. Notably, several examples showed a substantial decrease in verification time.

5.6 User-provided inverse functions

This section proposes a novel idea for reusing Viper's domain feature to define inverse functions for ISCs. Domains allow users to define types, mathematical functions, and axioms that establish their properties. More details about domains can be found in the Viper tutorial [5]. The proposed optimization leverages these user-provided domain definitions, particularly those resembling inverse functions, instead of generating new inverse function axioms.

5.6.1 Approach

One of the most common use cases for ISCs is to reason about arrays. Although Viper does not directly support arrays, they can be encoded via domains.

```
field val: Int
domain IArray {
    function loc(a: IArray, i: Int): Ref
    function len(a: IArray): Int
    function second(r: Ref): Int
    axiom all_diff {
        forall a: IArray, i: Int :: {loc(a, i)}
            second(loc(a, i)) == i
    }
    axiom length_nonneg {
        forall a: IArray :: len(a) >= 0
    }
}
```

Listing 5.10: Typical encoding of arrays in Viper via a domain.

Listing 5.10 shows three functions used to encode arrays. The functions `loc` and `len` denote element access and the overall length of the array, respectively. Additionally, there is an axiom constraining the length of an array to be nonnegative.

The function `second` encodes the uniqueness of element locations, ensuring that each element of an array maps to a distinct location. Crucially, all functions and axioms in a domain are translated directly to the SMT level. This means that we can reuse these functions and axioms.

```
method foo() {
    var a: IArray
    inhale forall i: Int :: {loc(a, i)}
        0  <= i && i < len(a) ==> acc(loc(a, i).val, write)
    exhale forall i: Int :: {loc(a, i)}
        0  <= i && i < len(a) ==> acc(loc(a, i).val, write)
}
```

Listing 5.11: Typical usage of an encoded array in Viper.

In Listing 5.11, we see two ISCs. As previously mentioned, reasoning about each ISC requires defining an inverse function and two corresponding definitional axioms. Our goal is to propose an alternative array encoding, similar

to the one shown in Listing 5.10, that allows us to avoid both the explicit definition of an inverse function and its associated axioms.

We divide the following explanation into two parts. First, we explain how to adapt the function `second` from the array encoding so that it can be reused as an inverse function. Second, we describe how to adapt the domain axioms to eliminate the need for separate definitional axioms for ISCs.

Let us begin with the first part. For now, we restrict the concept to receiver functions of the form `loc(a, i)`, where `i` is the quantified variable. For example, we disallow expressions such as `loc(a, f(i))`, where `f(i)` is an arbitrary function.

Given such an ISC for array a_j, we denote the corresponding receiver function as $e_{a_j}(i) := loc(a_j,\ i)$ and the corresponding condition function as $c_{a_j}(i)$. We denote the image of $e_{a_j}(i)$ as $Ref_{a_j} := \{e_{a_j}(i) \mid i \in Int \text{ and } c_{a_j}(i)\}$.

Since e_{a_j} is injective, an inverse function exists on its image. We denote this inverse function as $e_{a_j}^{-1} : Ref_{a_j} \to Int$. This inverse function must satisfy two constraints.

$$\forall i : Int.\ e_{a_j}^{-1}(e_{a_j}(i)) = i \tag{5.11}$$

$$\forall r : Ref_{a_j}.\ e_{a_j}(e_{a_j}^{-1}(r)) = r \tag{5.12}$$

We want to check if `second` satisfies these constraints. Clearly, Equation 5.11 holds for `second`. However, Equation 5.12 does not. The following proof demonstrates this.

Let us assume that Equation 5.12 holds for `second`. Now, consider an additional array a_k and suppose that some of a_k's locations overlap with a_j's locations, i.e. $\exists m, n \in Int.\ e_{a_j}(n) = e_{a_k}(m)$. Without loss of generality, we assume $n \neq m$. This allows us to derive the following equality:

$$\begin{aligned} e_{a_j}(n) &= e_{a_j}(second(e_{a_j}(n))) \\ &= e_{a_j}(second(e_{a_k}(m))) \\ &= e_{a_j}(m) \end{aligned} \tag{5.13}$$

The first equality follows from applying Equation 5.12, the second from the equality $a_j(n) = e_{a_k}(m)$, and the third again from Equation 5.12. The resulting equality, $e_{a_j}(n) = e_{a_j}(m)$, is clearly a contradiction since $m \neq n$. Therefore, our original assumption that Equation 5.12 holds for `second` must be false.

We can resolve this issue by modifying the definition of `second`. This is done by introducing an additional argument, making `second` dependent on the array. The updated definition is shown in Listing 5.12.

```
function second(a: IArray, r: Ref): Int
axiom all_diff {
    forall a: IArray, i: Int :: {loc(a, i)}
    second(a, loc(a, i)) == i
}
```

Listing 5.12: Updated definition of second.

With this new definition, we can avoid creating additional inverse functions. This concludes the first part of our explanation.

Next, we continue with the second part. The goal is to avoid adding any additional axioms for ISCs in our restricted form. We accomplish this by incorporating these axioms directly into the domain definition.

Let us first recall the definitional axioms for the inverse and image functions:

$$\forall x : \mathcal{T}.\ c(x) \implies img(e(x)) \wedge e^{-1}(e(x)) = x \tag{5.14}$$

$$\forall r : Ref.\ c(e^{-1}(r)) \wedge img(r) \implies e(e^{-1}(r)) = r \tag{5.15}$$

To fully replace those, we need to define an image function and add Equation 5.15 to the domain definition.

We begin by defining the image function such that it holds true whenever a given reference lies within the bounds of an array. For the same reason as the inverse function, the image function must also depend on the array.

```
function img(a: IArray, r: Ref): Bool
axiom imgDef {
    forall a: IArray, i: Int :: {loc(a, i)}
        0 <= i && i < len(a) ==> img(a, loc(a, i))
}
```

Listing 5.13: Image function definition via domains.

Finally, we state Equation 5.15 in terms of `second` and `img` and incorporate it into the domain definition.

```
axiom rightInverseAxiom {
    forall a: IArray, r: Ref :: {loc(a, second(a, r))}
        img(a, r) ==> loc(a, second(a, r)) == r
}
```

Listing 5.14: Equation 5.15 in terms of second and img.

Given these adaptations to the array domain definition, we can reuse this definition for ISCs and avoid introducing new axioms, as long as the receiver function remains in the form `loc(a, i)` and it can be proven that the condition function `c(i)` implies `img(a, loc(a, i))`. This concludes the concept of reusing user-provided functions as inverse functions.

As the next step, we extend this concept by slightly relaxing the allowed forms of receiver functions. Instead of allowing only expressions in the form `loc(a, i)`, we now also allow `loc(a, f(i))`, provided that `f` is bijective in `i` and its inverse is known.

Concretely, let $f(i)$ be defined as $f(i) := (+)(i,\ \mathit{offset})$, where *offset* is some constant. This function is bijective in i, and its inverse is $(-)(i,\ \mathit{offset})$. Using this, we can construct the required inverse function for such a receiver function via standard function composition:

$$(\mathit{loc} \circ (+))^{-1} = (+)^{-1} \circ \mathit{loc}^{-1} = (-) \circ \mathit{second} \tag{5.16}$$

This more general approach is important because defining domains in such a reusable way requires detailed knowledge of how these domain definitions interact at the SMT level, which a typical user might not have. However, Viper's frontends, such as Gobra [3], could be extended to apply this concept automatically, allowing end users to rely on the usual frontend syntax while still benefiting from a more efficient internal encoding.

Since slices are significantly more common than arrays in Go, the programming language targeted by Gobra, it would be beneficial to apply this concept to Gobra's slice encoding. A slice is a portion of an array, and its encoding can be seen as a wrapper around the basic array encoding plus an offset. In order to access an element of a slice, the offset (which indicates where the slice begins relative to the start of the array) is added to the array index. As a result, receiver functions take the form `loc(a, i + offset)`, which requires this more general approach.

In summary, this approach enables the reuse of user-defined functions from domains as inverse and image functions for ISCs. It can be applied to any ISC `inhale` or `exhale` where the condition and receiver match the form described above. Additionally, we allow the receivers to be composed with an invertible function for which an inverse is available. If the receiver is simple, that is, not composed with any additional functions, it is also sound to reuse user-defined functions if the ISC's condition implies the reused image function.

5.6.2 Experiments

This feature has not been fully implemented, so we conducted only preliminary manual tests to judge potential performance benefits. The experiments were performed on the Euler cluster at ETH Zurich [2], using a single CPU and 8 GB of RAM. Each benchmark was verified 30 times, and the results were averaged. This evaluation includes programs of increasing complexity as introduced in Section 4.6.1 with the exception of `manySets`, as this example does not contain arrays.

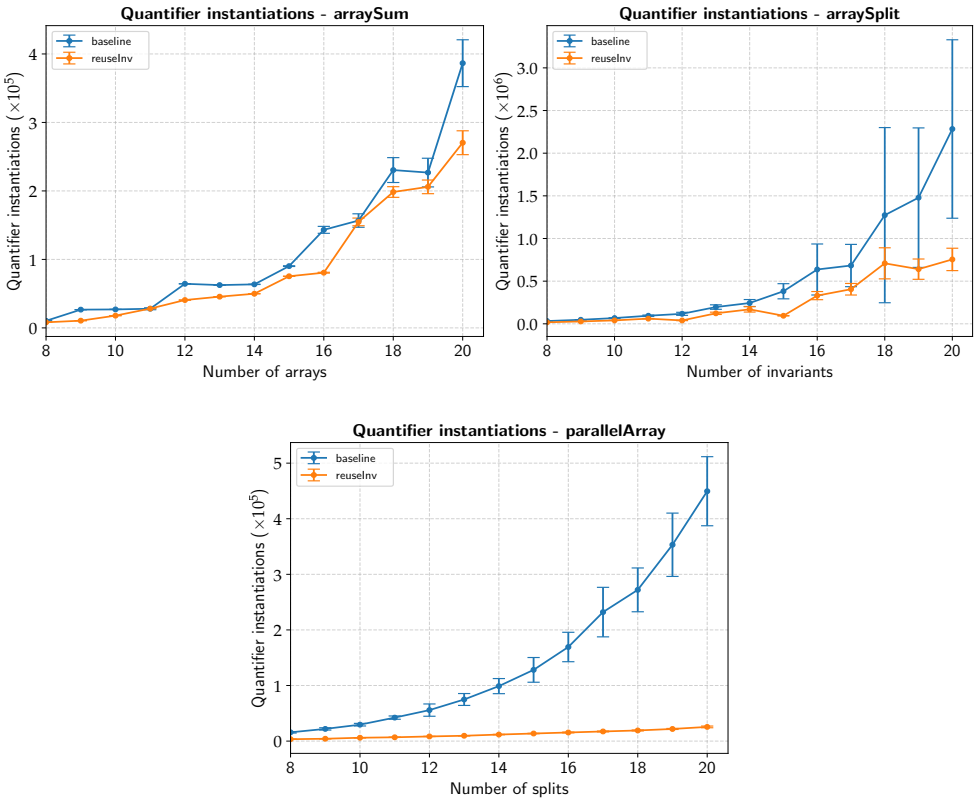

Figure 5.11: Number of quantifier instantiations for examples with increasing complexity. The X-axis represents the level of complexity, while the Y-axis shows the number of quantifier instantiations. Data points are averaged, and the standard deviation is included to indicate stability.

Figure 5.11 shows that this optimization reduces the number of quantifier instantiations across all three examples. Furthermore, all examples were successfully verified without requiring additional definitional axioms for ISC reasoning.

Next, we evaluate whether reducing quantifier instantiations translates into

faster overall verification times by examining the resulting speedups.

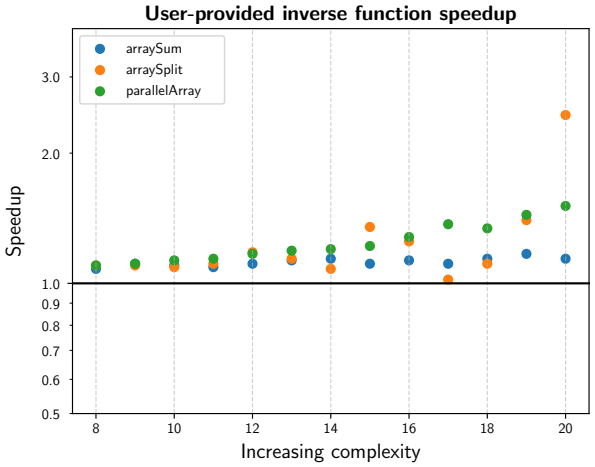

Figure 5.12: Speedup plot of examples. The X-axis represents the complexity level, while the Y-axis indicates the speedup. Data points represent the median verification time. The black line serves as the baseline reference at speedup = 1.

Although all of the examples in Figure 5.12 exhibit a speedup, the consistency of the results varies. Both `arraySum` and `parallelArray` show an exponential speedup. In contrast, `arraySplit` shows highly inconsistent behavior. This reflects the observations made in Section 5.4 on memoization where a similar instability was observed. Although the exact cause of this behavior in Z3 has not yet been identified, it is certainly worth investigating further.

As a final experiment, we investigated whether this approach is feasible in Gobra. Consider the Go function in Listing 5.15.

```
1 func foo() {
2     intSlice := make([]int, 5)
3     assert forall i int :: {&intSlice[i]}
4         0 <= i && i < len(intSlice) ==> acc(&intSlice[i])
5 }
```

Listing 5.15: Simple Gobra example involving slices.

The example in Listing 5.15 illustrates permission reasoning over Go slices. The program creates a slice of five integers using Go's `make` function. The `make` operation is encoded as an ISC that inhales full permissions for each element of the slice. This makes the example well-suited for testing whether the assertion can still be verified when the encoding relies on a reused domain definition.

We tested this by using Gobra to translate this Go code into Viper, then manually updated the domain definitions and included an additional `minus` function. We also manually defined the inverse and image functions that should be used for the ISCs. With this setup, we were able to verify the example successfully. That said, no general performance experiments were conducted due to the extensive manual labor required. However, we believe that these manual steps can be fully automated in the future.

Overall, these preliminary results indicate that reusing definitions and, thereby, simplifying proof obligations for the SMT solver leads to fewer quantifier instantiations and noticeable speedups. Since some instability was observed, further investigations should be conducted before fully implementing the approach. In our opinion, the potential benefits make it worthwhile to continue exploring this direction and to work towards a full implementation, both in Silicon and in frontends like Gobra.

Chapter 6

Conclusion

In this thesis, we introduced and evaluated several improvements to predicate handling and ISC verification in Viper, with a focus on optimizing Silicon. Our work addressed performance bottlenecks and proposed optimizations to balance completeness and efficiency.

Our key contributions include the at-most-n predicates feature, which limits predicate's permission amounts that can be held, and a greedy ISC approach, which offers a less heuristic-dependent method for handling ISCs in Silicon. The former simplifies predicate usage for both the user and the SMT solver, while the latter improves verification stability and retains the flexibility to fall back to the original algorithm when needed. Additionally, we extended state consolidation techniques in Silicon by introducing multiple heuristics aimed at mitigating the incompleteness introduced by the new greedy approach.

Beyond these main contributions, we also explored general optimizations designed to reduce the number of quantified axioms and quantifier instantiations, thereby simplifying the proof obligations for the underlying SMT solver. In addition, we proposed the novel concept of reusing user-defined inverse functions, an approach that leverages existing axioms to define inverse functions rather than generating new ones.

In summary, our proposed greedy ISC approach proves to be a viable alternative for ISC verification, offering a more stable solution while allowing to fallback to the previous algorithm if necessary. Our new predicate feature has shown to be useful in both Carbon and Silicon, enhancing control over permission handling. The introduced optimizations demonstrate that reducing quantifier instantiations and refining ISC handling can improve verification performance. While challenges remain, we believe that our contributions represent a helpful step forward in improving the efficiency and robustness of Viper.

6.1 Future work

In this section, we propose future work related to both the new predicate feature and the greedy approach. Additionally, we outline further potential ISC-related enhancements to Silicon.

- The at-most-n predicates feature, while useful, is currently limited in scope due to an injectivity issue in its proposed general approach. Future work could focus on developing an alternative method to overcome this limitation and further enhance the feature's capabilities.
- The current implementation of greedy read iterates chunks in a inefficient order, resulting in many avoidable SMT solver checks. We propose a syntactic similarity-based ordering that prioritizes chunk processing based on their receiver expressions. This approach may reduce the number of checks and improve the efficiency of the previously proposed greedy read strategy, which has shown suboptimal performance in its evaluation.
- State consolidation remains a challenge, especially in the presence of disjunctions where the new greedy algorithm still exhibits incompleteness. Future work could investigate alternative principles for determining when to consolidate states, aiming to improve overall completeness.
- The concept of reusing user-defined inverse functions was introduced and preliminarily tested but has not yet been fully implemented. Future work could focus on integrating this approach into both Carbon and Silicon. Furthermore, Viper's frontends, such as Gobra, would need to be adapted to provide the required definitions.
- Some quantified constraints differ only in the names of their quantified variables. It is unclear whether Z3 internally recognizes such equivalences. A potential improvement for Silicon would be to standardize quantified variable naming, preventing the generation of redundant axioms altogether.
- Lastly, the evaluation of some optimizations revealed unexpected behavior in Z3. In particular, memoization of inverse functions sometimes resulted in worse performance. A deeper investigation into Z3's internal heuristics and its handling of quantifier instantiations could offer valuable insights and guide future optimization efforts.

Bibliography

[1] Carbon. https://github.com/viperproject/carbon. Accessed on 2025-03-26.

[2] ETH euler cluster. https://scicomp.ethz.ch/wiki/Euler. Accessed on 2025-03-26.

[3] Gobra. https://www.pm.inf.ethz.ch/research/gobra.html. Accessed on 2025-03-26.

[4] SMT-lib. https://smt-lib.org/. Accessed on 2025-03-26.

[5] Viper tutorial. https://viper.ethz.ch/tutorial. Accessed on 2025-03-26.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. Technical report, ETH Zurich, 2019.

[7] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] Thibault Dardinier, Peter Müller, and Alexander J. Summers. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022.

[9] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[11] Lars. Issue #831: Multiple quantifiers to memory locations of same type. https://github.com/viperproject/silicon/issues/831, 2024. Accessed: 2025-03-26.

[12] Microsoft. Boggie verification language. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/. Accessed on 2025-03-26.

[13] Microsoft. Z3 prover. https://github.com/Z3Prover/. Accessed on 2025-03-26.

[14] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[15] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. *CoRR*, abs/1603.00649, 2016.

[16] J. C. Pereira, T. Klenze, S. Giampietro, M. Limbeck, D. Spiliopoulos, F. A. Wolf, M. Eilers, C. Sprenger, D. Basin, P. Müller, and A. Perrig. Protocols to code: Formal verification of a next-generation internet router. Technical report, 2024.

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

[18] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, 2016.

[19] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), May 2012.

[20] Alexander Stekelenburg. PR #860: Consolidating quantified field and predicate chunks. https://github.com/viperproject/silicon/pull/860, 2024. Accessed: 2025-03-26.

[21] Philippe Voinov. Optimisation of a deductive program verifier. Bachelor's Thesis, ETH Zurich, 2019.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

◉ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Enhancing Deductive Verification of Unbounded Heap Data Structures

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Limbeck	Markus

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zurich, 26.03.2025	Markus Limbeck

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard