Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automatic Inference of Hyperproperties

Bachelor's Thesis

Christian Knabenhans

August 23, 2018

Supervisors: Prof. Dr. Peter Müller, Jérôme Dohrau, Marco Eilers
Department of Computer Science, ETH Zurich

Contents

Chapter 1

Introduction

Today, automated reasoning about program behavior is a well-established discipline in computer science, with a wide array of tools and techniques. In the most common scenario, the goal is to prove trace properties of programs, such as termination or functional correctness. However, not all program properties can be expressed as properties of individual traces. *Hyperproperties* are properties which relate different executions of the same program, and include non-interference, determinism, and injectivity, among others. For example, proving determinism for a program requires showing that any two executions with identical initial states will produce identical final states. Non-interference classifies a program's inputs and outputs as either *low* (public) and *high* (secret). A program is said to fulfill non-interference if the low outputs are not influenced by the high inputs, i.e., if in several runs of the program with identical low inputs (but possibly different high inputs), the resulting low outputs are identical.

Eilers et al. developed a methodology to verify general hyperproperties (and non-interference in particular) [6], and implemented it in the Viper[1] [12] framework. To achieve this, they build a *modular product program*, which simulates several executions of the original program. A hyperproperty can then be expressed as a trace property of the product program, and verified using standard verifiers.

However, as is generally the case with deductive verification, Eilers' methodology requires specifications to be added to the program in the form of pre- and postconditions, and loop invariants. This is cumbersome in general, and specifications for hyperproperties are especially verbose and difficult to find.

This thesis aims to develop a framework to automatically infer such specifications, in order to prove hyperproperties with less or even without program-

[1] www.bitbucket.org/viperproject/silver

mer input. We develop a static analysis to infer specifications on product programs, and implement it in the static analyzer Sample[2,3], which is part of the Viper framework.

This report is organized as follows: in Chapter 2, we explain the concepts used in our methodology, in particular the construction of modular product programs. We detail our approach in Chapter 3, and discuss our implementation in Chapter 4. Finally, we evaluate our framework with respect to performance and precision in Chapter 5, and conclude in Chapter 6.

Throughout this report, we will explain our methodology for general hyperproperties, using non-interference as an example.

[2] www.pm.inf.ethz.ch/research/sample.html
[3] www.bitbucket.org/viperproject/sample

Chapter 2

Preliminaries

2.1 Hyperproperties

As mentioned in Chapter 1, a *hyperproperty* is a property that relates several executions of the same program. A *k-safety hyperproperty* relates finite prefixes of k execution traces, for an arbitrary k.

Interesting hyperproperties include:

- Non-Interference (NI): If a program is run twice with the same low inputs (but possibly different high inputs), its low outputs will be the same.
- Determinism: If a program is run twice with the same inputs, it will produce the same outputs in both executions. Determinism is equivalent to NI without high variables.
- Injectivity: If a program is run twice with different inputs, it will produce different outputs in both executions, i.e., the program implements an injective mapping from its input space to its output space.
- Symmetry: If a program with two inputs is run twice with swapped inputs, it will produce the same output in both executions.
- Transitivity: Suppose we have a program with two inputs and one boolean output. If the program outputs *true* for inputs a, b and b, c, it also outputs *true* for inputs a, c.

SIF, Determinism, Injectivity, and Symmetry are 2-safety hyperproperties whereas Transitivity is a 3-safety hyperproperty.

2.2 Product Program

2.2.1 Translating Statements

In order to infer k-safety hyperproperties on a program, we build a modular k-product program, as introduced by Eilers et al. [6]. A run of a k-product program corresponds to k runs of the original program, and we can thus express a k-safety property of some program as a trace property of its product program. The product program can then be fed to an off-the-shelf verifier or static analyzer.

We will now briefly explain how one can build a k-product program from a program.

The product program multiplies the state space of the original program by creating k renamed versions of all original variables. The product program also uses boolean *activation variables* that store, for each of the k executions, the condition under which it is active.

In the following, we will usually denote the i-th activation variable as $p^{(i)}$. We write $e^{(i)}$ for the i-th renamed version of the original expression e, and $\prod_k^{\mathring{p}}(s)$ for the k-product of the statement s parametrized by the activation variables $\mathring{p} \equiv p^{(1)}, \ldots, p^{(k)}$. We also let $\odot_{i=1}^{k} s_i$ denote the sequential composition of k statements $s_1\,;\ldots;s_k$.

Conditionals

For each *if-then-else*-statement in the original program, $2k$ new activation variables are created (k for each branch). The body of the branches are then translated, respectively parametrized by the new activation variables. More formally,

$$\begin{aligned}\textstyle\prod_k^{\mathring{p}}(\texttt{if}\ (c)\ \{s\}\ \texttt{else}\ \{s'\}) = {} & \odot_{i=1}^{k}\, t^{(i)} := p^{(i)} \land c^{(i)}\,; \\ & \odot_{i=1}^{k}\, f^{(i)} := p^{(i)} \land \neg c^{(i)}\,; \\ & \textstyle\prod_k^{\mathring{t}}(s)\ ;\ \prod_k^{\mathring{f}}(s').\end{aligned}$$

$\mathring{t}$ and $\mathring{f}$ are fresh variables, i.e., they do not occur anywhere else in the program.

Assignments

For every state-modifying statement (except assignments where the right-hand side is a method call, which are treated separately) in the original program, the product modifies the state of all active executions in the product program. This is achieved by creating k renamed statements from the original, and wrapping each of them in an if-statement with its activation variable as condition. For example, an assignment $x := e$ is translated to $\odot_{i=1}^{k}$ $\texttt{if(}p^{(i)}\texttt{)}\ \{x^{(i)} := e^{(i)}\}$.

Loops

Loops are not duplicated in the product program. Instead, a loop in the program is translated to a single loop in the product program, whose condition is the disjunction of the k renamed versions of the original loop condition, conjoined with their respective activation variables. This ensures that the loop body in the product is executed when at least one loop body is executed in an active execution. k new activation variables are created for the body; the i-th activation variable for the body is true iff the body is executed in the i-th execution. The body of the translated loop is the translated body of the original loop. More formally,

$$
\begin{aligned}
\prod\nolimits_k^{\mathring{p}} (\texttt{while(}e\texttt{)}\ \{s\}) = \texttt{while((}p^{(1)} \wedge e^{(1)}) \vee \ldots \vee (p^{(k)} \wedge e^{(k)})\texttt{)}\ \{& \\
\odot_{i=1}^{k}\ p_1^{(i)} := p^{(i)} \wedge e^{(i)}\texttt{;}& \\
\prod\nolimits_k^{\mathring{p}_1} (s)& \\
\}.&
\end{aligned}
$$

Procedure Calls

A procedure call in the original program is translated to a single call to the translated procedure in the product. The translated method takes k activation variables as arguments, as well as k renamed versions of every original argument. The activation variables passed as arguments are used to inform the callee about which executions are active.

The k renamed arguments are evaluated and stored in k temporary variables (conditionally on their respective activation variables). This ensures that arguments are only evaluated and passed on to the callee for active executions. The result of the translated call is then stored in k other temporary arguments, which are then copied to the k renamed targets, (conditionally on their respective activation variables). Again, this ensures that returned values are only evaluated and assigned to targets for valid executions.

These translated statements are then made conditional on the disjunction of all activation variables, to ensure that the procedure in the program is only called if at least one procedure in the original program is called. More formally,

$$\begin{aligned}\textstyle\prod_k^{\mathring{p}}(x_1,\dots,x_m := m(e_1,\dots,e_n)) = {} & \texttt{if } (p^{(1)} \vee \dots \vee p^{(k)})\ \{\\ & \quad \odot_{i=1}^{k} \texttt{if}(p^{(1)})\ \{\odot_{j=1}^{n}\ a_j^{(i)} := e_j^{(i)}\};\\ & \quad \mathring{t}_1,\dots,\mathring{t}_m := m(\mathring{p},\mathring{a}_1,\dots,\mathring{a}_n);\\ & \quad \odot_{i=1}^{k} \texttt{if}(p^{(1)})\ \{\odot_{j=1}^{m}\ x_j^{(i)} := t_j^{(i)}\}\\ & \}.\end{aligned}$$

Here $\mathring{t}_1,\dots,\mathring{t}_m,\mathring{a}_1,\dots,\mathring{a}_n$ do not occur anywhere else in the program.

2.2.2 Modeling the Heap

There are several ways to translate the heap in the product: we could duplicate all reference variables, but keep the original fields, or we could duplicate both reference variables and fields. For this thesis, we opt for the latter; this enables us to be more fine-grained. For example, we can express that r is the same in two executions (namely as $r^{(i)} = r^{(j)}$), even if the field access $r.f$ is different in these executions (i.e., $r^{(i)}.f^{(i)} \neq r^{(j)}.f^{(j)}$). If fields are not duplicated, $r^{(i)} = r^{(j)}$ implies $r^{(i)}.f = r^{(j)}.f$, i.e, $r.f$ can only have the same value in different executions if r points to the same heap location in these executions. In particular, it is not possible to specify the NI hyperproperty $low(r)$ without also specifying $low(r.f)$ using this encoding of the heap.

When duplicating the fields, an expression $r.f$ in the original program becomes $r^{(i)}.f^{(i)}$ for the i-th execution in the product program.

To translate an allocation $r := new(f_1,\dots,f_n)$ for a reference r and fields $f_1,\dots,f_n$, a (fresh) temporary variable tmp is created, allocated, and each of the k renamed versions of r are assigned tmp, conditionally on their respective activation variables. This construction ensures that newly allocated references are low by default, since the k references in the active executions of the product all point to tmp. Formally,

$$\begin{aligned}\textstyle\prod_k^{\mathring{p}}(r := \texttt{new}(f_1,\dots,f_n)) = {} & tmp := \texttt{new}(\mathring{f}_1,\dots,\mathring{f}_n);\\ & \odot_{i=1}^{k}\ r^{(i)} := tmp.\end{aligned}$$

2.3 Translating Specifications

To be able to reason about k-safety hyperproperties, we need to translate specifications from an original program to its product program, and to translate specifications inferred on the product program back to the original program.

We distinguish two types of specifications: *unary* specifications, which describe properties that should hold for each single execution (e.g. a precondition that guarantees the termination of the program), and *k-relational* specifications, which describe the relation between the states of k executions.

A unary specification in the product program should hold if the execution is active. Therefore, the transformation of a unary specification P is $\bigwedge_{i=1}^{k}(p^{(i)} \Rightarrow P^{(i)})$ in a k-product program with activation variables $p^{(1)}, \ldots, p^{(k)}$.

A k-relational specification, on the other hand, only makes sense when all k executions are active. Thus, the transformation of a k-relational specification $\hat{P}$ is $\bigwedge_{i=1}(p^{(i)}) \Rightarrow \hat{P}$ in a product program with activation variables $p^{(1)}, \ldots, p^{(k)}$.

We introduce a keyword *rel*; $rel(e, i)$ in a program is translated to $e^{(i)}$ in its k-product program. For the special case $k = 2$, we also introduce the keyword *low*, and translate $low(e)$ to $e^{(1)} = e^{(2)}$ in the product. *low* expressions are especially useful for specifying NI properties.

These two keywords allow us to specify k-relational properties we want to verify for a program, or in our case, to specify inferred k-relational properties as pre- and postconditions, as well as loop invariants of an original program.

2.4 Abstract Interpretation

Abstract Interpretation[1] (AI), introduced by Cousot and Cousot [2], is a mathematical framework for the (sound) approximation of the semantics of programs. A semantics is a mathematical characterization of all possible behaviors of a program. AI can be viewed as a partial execution which gains information about the semantics of a program without performing all the concrete calculations.

The most precise semantics is the *concrete semantics*, which consists of a set of traces a program may produce. A trace is a sequence of consecutive states, which contain information about the current program point, local variables, and heap. The goal of AI is to derive a (computable) semantic interpretation for each program point. Generally, AI trades off the precision of the analysis against its tractability.

Let $(C, \leq)$ and $(A, \sqsubseteq)$ be two partially ordered sets, let $\alpha : C \rightarrow A$ be the *abstraction function*, and let $\gamma : A \rightarrow C$ be the *concretization function*. C is the *concrete set* and A is the *abstract set*. For $\xi \in C$ and $x \in A$, $\alpha(\xi)$ is the *abstraction* of ξ, and $\gamma(x)$ is the *concretization* of x.

$\bot$ and $\top$ are the least and greatest elements in A, i.e., $\bot \sqsubseteq x \sqsubseteq \top$ for all $x \in A$. For $x, y \in A$, $\sqcup$ is the *least upper bound* operator, with $x \sqsubseteq x \sqcup y$ and $y \sqsubseteq x \sqcup y$, and $\sqcap$ is the *greatest lower bound* operator, with $x \sqcap y \sqsubseteq x$ and

[1] www.di.ens.fr/~cousot/AI/IntroAbsInt.html

$x \sqcap y \sqsubseteq y$. We assume that A is a complete lattice, i.e., that $\perp$ and $\top$ exist, and that $\sqcup$ and $\sqcap$ are defined for all elements of A.

The concrete semantics can be expressed as a fixed-point of a monotonic function $f : C \rightarrow C$. A *valid abstraction* $F : A \rightarrow A$ of f is a function such that $f(\gamma(x)) \leq \gamma(F(x))$ for all $x \in A$. Any x satisfying $F(x) \sqsubseteq x$ is then an abstraction of the least fixed point of f. The goal of AI is to compute such an x.

Such an abstraction of the least fixed-point can be computed as the stationary limit of the sequence x_n defined by $x_0 = \perp, x_{n+1} = F(x_n)$. In some cases, particularly when A is of infinite height, this sequence might not converge, but instead grow indefinitely. For these cases, x can be obtained through the widening operator $\triangledown$. $\triangledown$ should provide an upper bound of its arguments (i.e., $x, y \sqsubseteq x \triangledown y$), and for any sequence $\{y_i\}$, the sequence defined as $x_0 = \perp, x_{n+1} = x_n \triangledown y_n$ should converge to a stationary point.

Thus, AI abstracts sets of traces (elements of C) as abstract elements of A.

We now describe some well-known (numerical) abstract domains.

- **Sign:** An element of the sign domain is a mapping from local variables to their sign. As such, the sign domain only tracks information about individual variables.
- **Octagon:** [11] An element of the sign domain is a constraint of the form $\pm u \pm v \leq c$, where u and v are program variables and c is a constant. The octagon domain is thus a *relational domain*, i.e., it can capture information about the relationship between variables. However, only a subset of all relationships between variables can be captured. For example, the constraint $u = v \pm w$ cannot be expressed, and an imprecise over-approximation will be used instead.
- **Polyhedra:** [3] An element of the polyhedra domain stores constraints of the form $a_1 v_1 + \ldots + a_n v_n \leq c$ for constants $a_1, \ldots, a_n, c$ and variables $v_1, \ldots, v_n$. Therefore, the polyhedra domain can express linear relations between an arbitrary number of variables, and is a relational domain.

It is possible to combine two different abstract domains to a single one using a *reduced product* construction [4]. The idea behind a reduced product is to communicate the information from one domain to the other domain, and vice-versa, in order to obtain a more precise result than when using each domain in isolation. One particularly useful reduced product combines a heap domain, which abstracts information about the heap, with a numerical domain, which abstracts information about numerical properties of variables and fields.

Chapter 3

Approach

3.1 Motivation

As a motivating example, consider the simple Viper method `m` and its 2-product in Figure 3.1. `m` implements the ramp function, which returns 1 if its input is positive, and 0 otherwise. Our goal is to infer the NI postcondition $low(x) \Rightarrow low(res)$ on `m`, which specifies that its output is low if its input is low.

```
method m(x: Int)
       returns (res: Int)
{
  if (x > 0) { res := 1 }
  else { res := 0 }
}
```

```
method m(p_1: Int, p_2: Int,
         x_1: Int, x_2: Int)
       returns (res_1: Int,
                res_2: Int)
{
  var pt_1: Int
  var pt_2: Int
  var pf_1: Int
  var pf_2: Int
  pt_1 := p_1 && (x_1 > 0)
  pt_2 := p_2 && (x_2 > 0)
  pf_1 := p_1 && !(x_1 > 0)
  pf_2 := p_2 && !(x_2 > 0)
  /* s */
  if (pt_1) { res_1 := 1 /* s’ */}
  /* s’’ */
  if (pt_2) { res_2 := 1 }
  if (pf_1) { res_1 := 0 }
  if (pf_2) { res_2 := 0 }
  if (p_1) { ret_1 := 2 }
  if (p_2) { ret_2 := 2 }
}
```

Figure 3.1: A simple Viper method (left), and its 2-product (right).

As described in Section 2.3, we build a product program, on which we infer the postcondition $(p^{(1)} \wedge p^{(2)}) \Rightarrow x^{(1)} = x^{(2)} \Rightarrow res^{(1)} = res^{(2)}$. We then translate this inferred postcondition back to the original program, and obtain the desired postcondition.

We want to use AI to infer the postconditions of the product. Suppose we run a static analysis using the polyhedra domain on the product method. Let s, s' and s'' denote the abstract states before the if-statement in line 15, at the end of the body, and after the if-statement, respectively. s is $\top$: no information is known about any variable. The abstract state s' contains the constraint $res^{(1)} = 1$. However, this information is lost in $s'' = s \sqcup s' = \top \sqcup s' = \top$. This situation is repeated in the conditionals of lines 17–21, and the abstract state at the end of the method will not contain any information about the return variables.

This loss in precision stems from the structure of the product program itself; by construction of the product, the two branches of a conditional are ripped apart, and the control flow is made dependent on the activation variables. This "implicit" control flow is not understood by standard static analyses, which leads to precision losses, as can be seen in the example above. Additionally, a product combines the control flows of k different executions, which poses an additional challenge to the analysis. To gain some precision, we need to somehow "restore" the original control flows, and make them explicit for the static analysis. We do this using *trace partitioning*.

3.2 Trace Partitioning

Usually, a static analysis approximate sets of program traces. Trace partitioning [10] allows one to get a more precise abstraction by performing the abstraction over a partition of the set of traces instead of the set itself.

In our case, we want to partition the set of traces on whether the i-th execution is active (for each program point). A natural way to achieve this partition is to distinguish traces in which the i-th activation variable is true from those in which it is false.

For example, in the product method from Figure 3.1, we would split the set of traces into $2^4 = 16$ partitions, based on whether `pt_1`, `pt_2`, `pf_1`, and `pf_2` are true or false. Abstracting each of these partitions separately, our simple analysis from before knows, at the end of the method, that the return variables `res_1` and `res_2` are equal in the traces where only `pt_1` and `pt_2`, or only `pf_1` and `pf_2` are true. For all other traces, the analysis either knows that the return variables are not equal, or knows nothing about the relationship between them.

These partitions evidently help the analysis to gain some precision, but we are still unable to infer NI specifications from the abstract states (e.g., we only know that the return variables are equal if only `pt_1` and `pt_2` are true).

To bridge the gap to NI specifications inference, we introduce an additional partition of the traces, namely on whether x is low in the original program. An input x is low in the original program if $x^{(1)} = x^{(2)}$ in the product. With this new partitioning expression, we obtain $2^5 = 32$ partitions of the set of traces. Some of these partitions are empty, e.g., there is no valid trace in which `pt_1` and `pf_1` are both true, and there is no valid trace in which $x^{(1)} = x^{(2)}$, `pt_1`, and `pf_2` are true.

Finally, since we only want to infer specifications on the product that can be transformed to k-safety hyperproperties of the original program, we are only interested in properties of the form $(\bigwedge_{i=1}^{k} p^{(i)}) \Rightarrow P$. Therefore, we can assume that $p^{(1)}, \dots, p^{(k)}$ are true to further increase precision (e.g., there is no valid trace where $p^{(1)}$, $p^{(2)}$, $x^{(1)} = x^{(2)}$ and `pt_1` are true, but `pt_2` is false).

Using the partitions described above, the simple analysis used for `m` in Figure 3.1 can infer that the return variables are equal in all traces in which $x^{(1)} = x^{(2)}$, under the assumption that $p^{(1)}$ and $p^{(2)}$ are true, i.e., that $p^{(1)} \wedge p^{(2)} \Rightarrow x^{(1)} = x^{(2)} \Rightarrow res^{(1)} = res^{(2)}$.

We describe how we inform our abstract domain about the partitions in Chapter 4.

3.3 Abstract Domain

3.3.1 Binary Decision Trees

To implement trace partitioning, we use a Binary Decision Tree (BDT) abstract domain [1]. As its name suggests, a BDT domain partitions traces using a binary tree, where nodes store boolean expressions. The left and right subtrees T and F of a node $[\![\, c : T, F \,]\!]$ with condition c abstract sets of concrete states, reachable via traces for which c and $\neg c$ hold, respectively. The domain takes another abstract domain as parameter (the leaf abstract domain $\mathbb{L}$), and the leaves (written as $(\!|\cdot|\!)$) of the tree store constraints as elements of $\mathbb{L}$.

For some leaf abstract domain $(\mathbb{L}, \bot_{\mathbb{L}}, \top_{\mathbb{L}}, \sqsubseteq_{\mathbb{L}}, \sqcup_{\mathbb{L}}, \sqcap_{\mathbb{L}}, \triangledown_{\mathbb{L}})$, we denote the BDT domain parametrized by $\mathbb{L}$ as $\mathbb{T}_{\mathbb{L}}$. The elements BDT of $\mathbb{T}_{\mathbb{L}}$ are defined by the following syntax:

$$\begin{aligned} BDT &::= \bot \mid \top \mid Inner \\ Inner &::= (\!| S |\!) \mid [\![\, B : Inner, Inner \,]\!] \end{aligned}$$

Here $S \in \mathbb{L}$ is an element of the leaf abstract domain, and B is a boolean expression.

3.3.2 Operators

The abstract domain operators $\sqsubseteq$, $\sqcup$, $\sqcap$, and $\triangledown$ are defined as follows:

Inclusion

$$a \sqsubseteq a' = \begin{cases} s \sqsubseteq_{\mathbb{L}} s' & \text{if } a = (\!| s |\!) \text{ and } a' = (\!| s' |\!) \\ T \sqsubseteq T' \wedge F \sqsubseteq F' & \text{if } a = [\![c : T, F]\!] \text{ and } a' = [\![c : T', F']\!] \\ false & \text{otherwise} \end{cases}$$

Least Upper Bound

$$a \sqcup^{B} a' = \begin{cases} (\!| (s \sqcup_{\mathbb{L}} s') \sqcap_{\mathbb{L}} \alpha_{\mathbb{L}} (\bigwedge_{b \in B} b) |\!) & \text{if } a = (\!| s |\!) \text{ and } a' = (\!| s' |\!) \\ [\![c : T \sqcup^{B \cup \{c\}} T', F \sqcup^{B \cup \{\neg c\}} F']\!] & \text{if } a = [\![c : T, F]\!] \text{ and } a' = [\![c : T', F']\!] \\ \top & \text{otherwise} \end{cases}$$

Here $\sqcup \equiv \sqcup^{\{\}}$, and $\alpha_{\mathbb{L}}$ is a function mapping boolean expressions to their abstraction in the leaf domain $\mathbb{L}$. The bound B is necessary to ensure that c (respectively $\neg c$) still holds in the left (respectively right) child of a node with condition c in the resulting BDT.

Greatest Lower Bound

$$a \sqcap a' = \begin{cases} (\!| s \sqcap_{\mathbb{L}} s' |\!) & \text{if } a = (\!| s |\!) \text{ and } a' = (\!| s' |\!) \\ [\![c : T \sqcap T', F \sqcap F']\!] & \text{if } a = [\![c : T, F]\!] \text{ and } a' = [\![c : T', F']\!] \\ \bot & \text{otherwise} \end{cases}$$

Widening

The widening operator $\triangledown$ is defined similarly to the least upper bound operator.

3.3.3 Splits and Merges

We also introduce two new operators: *split* and *merge*. A formal definition is shown in Figure 3.2.

$split(T, c)$ replaces every leaf in T by a node with condition c, such that c and $\neg c$ hold in its left and right children, respectively. In other words, *split* adds a new partition (on c) of the traces.

merge(T, c) replaces every node with condition c in T by the least upper bound of its children, i.e., it returns a tree where the partition on c has been forgotten.

$$split(\llparenthesis\, s \,\rrparenthesis,\ c) = \llbracket\, c : s \sqcap_{\mathbb{L}} \alpha_{\mathbb{L}}(c), s \sqcap_{\mathbb{L}} \alpha_{\mathbb{L}}(\neg c) \,\rrbracket$$
$$split(\llbracket\, c : T, F \,\rrbracket,\ c') = \begin{cases} \llbracket\, c : T, F \,\rrbracket & \text{if } c = c' \\ \llbracket\, c : split(T, c'), split(F, c') \,\rrbracket & \text{otherwise} \end{cases}$$

$$merge(\llparenthesis\, s \,\rrparenthesis,\ c) = \llparenthesis\, s \,\rrparenthesis$$
$$merge(\llbracket\, c : T, F \,\rrbracket,\ c') = \begin{cases} T \sqcup F & \text{if } c = c' \\ \llbracket\, c : merge(T, c'), merge(F, c') \,\rrbracket & \text{otherwise} \end{cases}$$

Figure 3.2: Definition of the *split* and *merge* operators on the BDT domain.

3.4 Storeless Heap Domain

3.4.1 Motivation

In general, we wish to infer hyperproperty specifications for heap-manipulating programs. For example, we may want to infer the specification $low(r.f)$ for some reference r with field f. To achieve this, we need to track information about the heap, in addition to numerical constraints. For heap-manipulating programs, the leaf domain of our BDT domain is therefore the reduced product of a heap abstract domain with a numerical domain.

An element of a simple heap abstract domain stores two mappings h and s. h maps a reference variable to a set of abstract objects, and s maps an abstract object and a field to a set of abstract objects. An abstract object represents a fraction of the entire heap (i.e., multiple concrete heap locations): for example, one could define an abstract object for each allocation site (i.e., for each program point where an allocation occurs).

A reference variable r can then point to any abstract object in $h(r)$, and a field access $r.f$ can point to any abstract object in $\bigcup_{O \in h(r)} s(O, f)$.

Unfortunately, such a simple heap domain suffers from imprecisions. Suppose a program contains three variables p, q, and r, and suppose there are three abstract objects O_1, O_2, and O_3. For the sake of simplicity, we assume that each abstract object only represents a single variable, i.e., if two references point to the same abstract object, they are equal.

Consider the situation shown in Figure 3.3. In this abstract state of a simple heap domain, $h(p) = h(q) = \{O_1\}$, $h(r) = \{O_2\}$, and $s(O_1, f) = \{O_3\}$. After executing $q.f := r$, $s(O_1, f)$ is weakly updated to $\{O_2, O_3\}$, since $p.f$ could still point to O_3. If the statement $r := q$ is now executed, $h(r)$ is

$$\begin{tikzcd}
p \arrow[dr] & O_3 & & p \arrow[dr] & O_3 & & p \arrow[dr] & O_3 \\
q \arrow[r] & O_1 \arrow[u, "f"'] & & q \arrow[r] & O_1 \arrow[u, "f"'] \arrow[d, "f"'] & & q \arrow[r] & O_1 \arrow[u, "f"'] \arrow[d, "f"'] \\
r \arrow[r] & O_2 & & r \arrow[r] & O_2 & & r \arrow[ur] & O_2
\end{tikzcd}$$

$\xrightarrow{q.f := r}$ (a) to (b), $\xrightarrow{r := q}$ (b) to (c)

(a)	(b)	(c)
$p \mapsto \{O_1\}$	$p \mapsto \{O_1\}$	$p \mapsto \{O_1\}$
$q \mapsto \{O_1\}$	$q \mapsto \{O_1\}$	$q \mapsto \{O_1\}$
$r \mapsto \{O_2\}$	$r \mapsto \{O_2\}$	$r \mapsto \{O_1\}$
$p.f \mapsto \{O_3\}$	$p.f \mapsto \{O_3\}$	$p.f \mapsto \{O_3\}$
$q.f \mapsto \{O_3\}$	$q.f \mapsto \{O_2\}$	$q.f \mapsto \{O_2\}$
		$r.f \mapsto \{O_2\}$

Figure 3.3: Points-to graphs for a simple heap domain (top) and points-to information for a storeless heap domain (bottom). For $s_1 \equiv q.f := r$, $s_2 \equiv r := q$, the abstract state is shown (a) before s_1, (b) after s_1 and before s_2, (c) after s_1 and s_2. In a points-to graph, there is an edge from r to O iff $O \in h(r)$, and there is an f-edge from O to O' iff $O' \in s(O, f)$.

updated to $\{O_1\}$. In the resulting abstract state, we cannot say with certainty that $q.f = r.f$, because according to the abstraction, it could be that $q.f$ is pointing to O_2 and $r.f$ to O_3.

Since we want to be able to express equalities between field accesses, a simple heap domain is not suitable for our analysis. To alleviate this loss in precision, we use a *storeless heap domain* [5]. An element of a storeless heap domain is a mapping m from *access paths* (i.e., expressions of the form $o.f_1 \cdots .f_n$ for a reference o and fields $f_1, \ldots, f_n$) to sets of abstract objects.

The abstract state for the situation we described above is shown in Figure 3.3, on the right. After the assignment $q.f := r$, $m(q.f)$ is (strongly) updated to O_2, and after $q := p$, $m(q)$ is updated to O_1. After these two statements have been executed, $q.f$ and $r.f$ both point to only one abstract object O_2, and we know that $q.f = r.f$.

We implemented such a storeless heap domain in Sample, and used it to infer hyperproperties for heap-manipulating programs.

3.5 Instantiations

As we showed in Section 3.2, we use trace partitioning to restore the original control flows from the product program. We also partition the traces w.r.t. interesting properties of the inputs.

For example, to infer NI specifications for a method with parameters $a_1, \dots, a_n$, we partition the traces on whether $low(a_1), \dots, low(a_n)$ holds. This is equivalent to running the analysis 2^n times, and assuming in each run that all variables in a subset of the arguments are low. We can then infer specifications of the form $low(a_i) \Rightarrow \dots \Rightarrow low(a_j) \Rightarrow low(e)$ for some expression e.

Property	**Partitioning Expressions**	**Example Hyperproperty**
SIF	$a^{(1)} = a^{(2)}, \dots, z^{(1)} = z^{(2)}$	$low(\vec{a}) \Rightarrow low(res)$
Determinism		$\vec{a} = \vec{a}' \Rightarrow m(\vec{a}) = m(\vec{a}')$
Monotonicity	$a^{(1)} \star a^{(2)}$ for $\star \in \{\leq, =, \geq, \dots\}$	$a \star a' \Rightarrow m(a) \star m(a')$
Injectivity	$\vec{a}^{(1)} \neq \vec{a}^{(2)}$	$\vec{a} \neq \vec{a}' \Rightarrow m(\vec{a}) \neq m(\vec{a}')$
Symmetry	$a^{(1)} = b^{(2)} \wedge a^{(2)} = b^{(1)}$	$R(a,b) \Leftrightarrow R(b,a)$
Asymmetry		$R(a,b) \Rightarrow \neg R(b,a)$
Total Relation		$R(a,b) \vee R(b,a)$
Anti-symmetry	$a^{(1)} = b^{(2)} \wedge a^{(2)} = b^{(1)} \wedge a^{(1)} \neq b^{(1)}$	$a \neq b \Rightarrow \neg(R(a,b) \wedge R(b,a))$
Transitivity	$a^{(1)} = a^{(3)} \wedge b^{(1)} = a^{(2)} \wedge b^{(2)} = b^{(3)}$	$(R(a,b) \wedge R(b,c)) \Rightarrow R(a,c)$

Table 3.1: An overview of hyperproperty specifications and the partitioning expressions used to infer them.

A few examples of hyperproperty specifications and initial partitioning expressions are given in Table 3.1. m is a method with arguments $\vec{a} \equiv a, \dots, z$ and return value res, and R is a method with arguments a and b, which returns a boolean.

3.6 Specification Inference

We will now describe how specifications can be obtained from a BDT. To this end, we define a function $\mathcal{C}^{\mathcal{L},\mathcal{R}}$, parametrized by sets of variables $\mathcal{L}$ and $\mathcal{R}$, which maps a BDT to a set of boolean constraints. We construct $\mathcal{C}^{\mathcal{L},\mathcal{R}}$ such that all expressions in the set of constraints are of the form $l_1 \Rightarrow \dots \Rightarrow l_n \Rightarrow r$, where l_i and r are expressions which only contain variables from $\mathcal{L}$ and $\mathcal{R}$, respectively. This constraint is equivalent to $(l_1 \wedge \dots \wedge l_n) \Rightarrow r$. We assume we are given a function $\mathcal{C}_{\mathbb{L}}^{\mathcal{V}}$ which maps elements of the leaf domain $\mathbb{L}$ to sets of constraints which only contain variables from $\mathcal{V}$. A formal definition of $\mathcal{C}^{\mathcal{L},\mathcal{R}}$ is given in figure 3.4.

$\mathcal{C}^{\mathcal{L},\mathcal{R}}$ is parametrized by two sets of variables in order to only keep specifications that can be translated back to the original program. In particular, these constraints should not contain activation variables (except those passed as arguments to the current method), since they are an artifact of the product construction and do not occur in the original program.

$$\begin{aligned}
&\mathcal{C}^{\mathcal{L},\mathcal{R}}(\top) = \{\} \\
&\mathcal{C}^{\mathcal{L},\mathcal{R}}(\bot) = \{false\} \\
&\mathcal{C}^{\mathcal{L},\mathcal{R}}(\llparenthesis\, s \,\rrparenthesis) = \mathcal{C}^{\mathcal{R}}_{\mathbb{L}}(s) \\
&\mathcal{C}^{\mathcal{L},\mathcal{R}}(\llbracket\, c : T, F \,\rrbracket) = \begin{cases} \quad\{c \Rightarrow \gamma \mid \gamma \in \mathcal{C}^{\mathcal{L},\mathcal{R}}(T)\} \\ \cup \{\neg c \Rightarrow \gamma \mid \gamma \in \mathcal{C}^{\mathcal{L},\mathcal{R}}(F)\} & \text{if } c \text{ only contains variables from } \mathcal{L} \\ \mathcal{C}^{\mathcal{L},\mathcal{R}}(T \sqcup F) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.4: Definition of the constraint inference function $\mathcal{C}^{\mathcal{L},\mathcal{R}}$.

For example, if we want to infer postconditions, we will choose $\mathcal{L}$ as the set of arguments of the product method, and $\mathcal{R}$ as the set of its return variables. This ensures that only constraints that can be translated to k-relational specifications are inferred.

Using $\mathcal{C}^{\mathcal{L},\mathcal{R}}$, we obtain a set of unary specifications of the k-product program, which we can then translate to k-safety properties of the original program by inverting the transformation described in section 2.3. Thanks to the structure of our inferred specifications, this translation can be done easily.

Chapter 4

Implementation

4.1 Splits

As we described in Chapter 3, we use trace partitioning to increase the precision of our abstract domain.

Obviously, a partition on an activation variable does not provide additional precision for the program points before the first occurrence of said variable.

Taking advantage of the well-defined structure of product programs, we can statically determine at which program point and on which condition we want to partition traces. To this end, we introduce a new Viper keyword `split`, which takes three boolean expressions as arguments. A statement `split(`c`, `t`, `f`)` informs the static analysis that it should partition traces on whether c is true or false, and that it can assume that t and f hold in the traces where c is true and false, respectively.

We instrument the product program by inserting splits, in order to inform the abstract BDT domain at which point and on which condition it should partition traces. We always insert splits after activation variables have been declared and initialized: for each assignment to a new activation variable `p_new := p && `c, we insert a statement `split(p_new, p && `c`, !(p && `c`))`.

We use "helper" conditions t and f because the simple leaf abstract domains that we use (e.g., octagon or polyhedra domains) are not precise enough to represent constraints of the form $p' = p \wedge c$. To ensure that we can still use information about p and c in the subtrees of the node with condition p', we communicate this information explicitly. This can be done, since p' is only assigned to once in the entire program (by construction of the product).

As an example, Figure 4.1 shows the product method from Figure 3.1 extended with `split` statements.

```
method m(p_1: Int, p_2: Int, x_1: Int, x_2: Int)
       returns (res_1: Int, res_2: Int)
{
  var pt_1: Int
  var pt_2: Int
  var pf_1: Int
  var pf_2: Int
  pt_1 := p_1 && x_1 > 0
  pt_2 := p_2 && x_2 > 0
  pf_1 := p_1 && !(x_1 > 0)
  pf_2 := p_2 && !(x_2 > 0)
  split(pt_1, p_1 && x_1 > 0, !(p_1 && x_1 > 0))
  split(pt_2, p_2 && x_2 > 0, !(p_2 && x_2 > 0))
  split(pf_1, p_1 && !(x_1 > 0), !(p_1 && !(x_1 > 0)))
  split(pf_2, p_1 && !(x_1 > 0), !(p_1 && !(x_1 > 0)))
  if (pt_1) { res_1 := 1 }
  if (pt_2) { res_2 := 1 }
  if (pf_1) { res_1 := 0 }
  if (pf_2) { res_2 := 0 }
}
```

Figure 4.1: The extended 2-product of the method `m` from fig. 3.1.

4.2 Merges

When we split on an activation variable p, we gain some precision in the abstract states for the program points after the split. In some cases, this gain in precision is not relevant for the properties we wish to infer, in particular for program points after the last occurrence of p. For such cases, it is useful to "forget" a partition, in order to keep the set of partitions (and therefore the size of the abstract state) reasonably small.

For this purpose, we introduce a second Viper keyword `merge`, which takes a single boolean expression as argument. A `merge`(c) statement informs the analysis that the partition of the traces w.r.t. the boolean condition c can be forgotten. Since the abstract states' sizes and the information they contain are not known before the static analysis is actually run, we do not know an optimal placement of `merge` statements beforehand. We can, however, use some heuristics.

We distinguish four `merge` strategies (referred to as M0–M3):

M0: One possible strategy is to never merge anywhere. This guarantees that no precision is lost, at the expense of efficiency.

However, due to the potentially exponential increase of the number of partitions, this strategy is rarely of practical interest.

For the three following strategies, at the end of a loop body, we merge all the unmerged splits in the body (which include the splits on the new activation variables for the body). We do this because we make the assumption that the splits inside the body are only useful for the body's statements; the activation variables for the body are not used later in the program. As will be shown in Chapter 5, we have found experimentally that this assumption typically holds. This assumption enables us to forget about at least k partitions after every loop, which helps to speed up our analysis.

M1: This strategy always merges on activation variables after their last occurrence in the program. This approach leads to precision losses, however in practice it is sufficient to infer specifications for many examples. This strategy essentially restores the branching factor in the original program.

M2: this strategy generates fewer unmerged splits than M3 and is more precise than M1. For each if-statement in the original program, the product contains $2k$ assignments to new activation variables, followed by $2k$ splits on these activation variables. We let C denote the set of these splits. After the splits come the $2k$ (or k, if there is no else-branch in the original program) translated versions of the bodies of the original if-statement, parametrized by the new activation variables.

After these translated bodies, we compute the set U of all unmerged splits up to this point (without the activation variables of the method), and the set R of all the remaining splits after this point (again, without the current method's activation variables). Writing V_S for the set of all variables occurring in the arguments of the splits in S (without the method's activation variables), we insert merges using the following rules:

- If $V_C \cap V_R \neq \{\}$, i.e., if the current splits have variables in common with future splits, we do not insert a merge command.
- If $V_C \cap V_R = \{\}$, i.e., if the current splits have no common variables with any future splits, we insert $2k$ merges corresponding to the splits in C. If V_C and V_R do not intersect, it is very probable that the partitions in C will not be useful for the remaining program points, and can thus be forgotten.

  Additionally, for each unmerged u = `split(`c`, `t`, `f`)` in U, we insert a corresponding `merge(`c`)` if $V_{\{u\}} \cap V_R = \{\}$, i.e., if u only has common variables with the current splits in C. $V_{\{u\}} \cap V_R = \{\}$ implies that u was not merged before because it was thought to be useful for the splits in C. Since if $V_C \cap V_R = \{\}$ the splits in C have just been merged, u can be merged as well.

Using this approach, we keep all partitions around up until the point at which we think they will not yield any precision in the future anymore, where we then forget about them.

However, this strategy fails when there are hidden data dependencies between variables. For example, consider the following statements:

```
y := 0; if (x > 0) { y := 1 }; if (y == 0) {...}
```

In the body of the second if-statement, the value of y depends on the value of x. However, since x does not appear in the second if-condition, we will merge on the activation variables of the first if-statement before splitting on those of the second, and therefore potentially lose precision.

It is possible to palliate this precision loss by running a simple data flow analysis on the original product to detect dependencies between variables, and using its results to decide where to insert merges. Nevertheless, we did not implement such a pre-analysis in our framework.

M3: Our last strategy is to only merge at the end of a loop body, as described above, and nowhere else. This guarantees that precision is only lost at the end of loop bodies.

For each of the three strategies M1, M2, and M3, we now give bounds for the number of open splits in a given sequence of statements.

For a given program point, we decompose the nesting level N as $N = I + L$, where I and L are the number of nested if-then-else-statements and loops at this program point, respectively. We let P denote the total number of previous if-then-else-statements in the program before this point. This includes the conditionals already counted by I.

Using the M1 strategy, only the splits on the activation variables of the I nested ifs and L nested loops are still unmerged. There are $2k$ and k activation variables for each if-then-else and loop, respectively. Therefore, there are exactly $(2k)^I k^L$ open splits at the program point.

The splits that are still open when using the M1 strategy are also open when using the M3 strategy. However, all the splits for previous if-then-else statements are still unmerged. Splits for previous loops and inside their bodies are merged. Thus, there are exactly $(2k)^I k^L (2k)^{(P-I)} = (2k)^P k^L$ open splits at the program point when using M3.

Since the M2 strategy depends on the program, we cannot give an exact bound for the number of unmerged splits at the program point. However, in the best case, M2 is equivalent to M1, and in the worst case, it is equivalent to M3. Therefore, there are between $(2k)^I k^L$ and $(2k)^P k^L$ unmerged splits at the current program point.

These bounds only include the splits on activation variables, which can be merged. To obtain the number of total open splits, one has add the number of initial splits on inputs.

The total number of splits is the height of the BDT in the abstract state; for S open splits at some program point, the corresponding BDT has 2^S leaves. Fortunately, a sizable portion of these leaves are $\bot_{\mathbb{L}}$. For example, an if-then-else-statement in the original product generates $2k$ splits on activation variables $t_1, \ldots, t_k, f_1, \ldots, f_k$.

For each pair of split t_i, f_i however, there is no valid trace where t_i and f_i are both true, independently of any assumption on inputs. Thus, these two splits only multiply the number of leaves by 3, and not by 4. After the $2k$ splits, the number of non-bottom leaves in the BDT as been multiplied by at most 3^k. When assumptions are made on inputs and input activation variables, this factor decreases further.

4.3 Property-dependent splits

We also use `splits` to express assumptions we make. For an assumption a, we insert a `split(`a`, `a`, false)` at the beginning of the method body. For example, as we are only interested in inferring k-relational specifications of the original program, we always make the assumption $p^{(1)} \wedge \ldots \wedge p^{(k)}$. These assumptions are actually program invariants that we assume: since they are located at the root of the BDT, they can be used for the entire subtree, for the entire duration of the analysis.

4.3.1 Parallelism

Taking advantage of the tree structure of our abstract states, we parallelize the operators $\sqcup$, $\sqcap$ and $\triangledown$ in our implementation. For this purpose, we use two thresholds T_{node} and T_{child}.

For any node $[\![c : T, F]\!]$, if the number of non-bottom leaves is above T_{node}, and if the numbers of non-bottom leaves of T and F are above T_{child}, we compute the results from T and F in two different threads, taken from a global thread-pool. Otherwise, we compute them sequentially. These two threads can, if the number of leaves is sufficiently high, delegate their computation again for the subtrees.

T_{child} ensures that we delegate work only if the tree is more or less balanced, i.e., if approximately as much effort is needed to compute the result for T and F. Typically, we used $T_{\text{node}} = 8$ or 16, and $T_{\text{child}} = \frac{1}{3} T_{\text{node}}$.

4.3.2 Explicit Bounds

When splitting using $split(c, t, f)$, t and f are assumed in the left and right subtrees, but not kept any longer. To gain some precision, we keep the expressions t and f stored at the nodes until they are invalidated, i.e., until an identifier in t or f is assigned to. By storing these helper assumptions, we can conjoin t and f with the node condition c, and assume it in the node's children before each operation, thereby gaining some precision.

This is useful if t or f are constraints which cannot be represented in the leaf abstract domain. For example, a constraint $x \neq 0$ cannot be represented in an abstract domain without disjunctions (which is the case for the octagon or polyhedra domains, for example).

Supposing we have a method parameter x, after a `split(q, p && x == 0, p && !(x == 0))`, we can keep the constraints $p, x = 0$ and $p, x \neq 0$ stored for the left and right subtree of the newly created node, respectively. Since p cannot be assigned again (by construction of the product), and since x cannot be assigned to (because of the syntax of Viper), $x \neq 0$ can be assumed for the right subtree for the entire analysis.

Chapter 5

Evaluation

We tested and evaluated our framework on examples collected from the literature, or that we created ourselves. The programs we wrote showcase important aspects of our approach. We chose programs from the literature on which the inference of hyperproperties presents a challenge; for example, programs containing loops, manipulating the heap, or on which standard approaches of hyperproperty inference are too imprecise to yield useful results.

For each example, we present the code, the inferred specifications (postconditions and loop invariants) for different leaf domains and for each of the merge placement strategies M1, M2, and M3 defined in Section 4.2. We also show the running times for each combination of domain and merge strategy.

In the following, Oct and Poly denote the octagon and polyhedra abstract domains, and H$\times$Oct and H$\times$Poly denote the reduced product of the storeless heap domain with the octagon and polyhedra domains, respectively. For Oct and H$\times$Oct, we use the implementation of the octagon domain present in Sample, which is written in Scala. For Poly and H$\times$Poly, we use the implementation of the polyhedra domain provided by the Apron library [8].

All running times presented in this chapter were measured on a Microsoft Surface Pro 3 (Intel i5 dualcore, 8GB of RAM) running Ubuntu 16.04, and averaged over 10 runs. Since our framework is written in Scala (which runs on the Java Virtual Machine), we ran the analysis 10 times before our measuring runs, to allow the JVM to gather profiling information and begin JIT compilation.

The rest of this chapter is organized as follows: in Section 5.1, we evaluate the inference of NI specifications on a corpus of examples. In Section 5.2, we evaluate the inference of Symmetry-like hyperproperties, and in Section 5.3, we infer Symmetry, Reflexivity and Monotonicity hyperproperties. Finally, we discuss the results of our evaluation in Section 5.4.

5.1 Non-Interference

The examples in this section are analyzed in order to infer NI specifications. The code and inferred specifications are shown for each example separately, and the timings for all the examples are shown together at the end of the section.

5.1.1 Postconditions and Loop Invariants

BA1

An existing approach to infer NI specifications is *taint analysis*. A taint analysis keeps track of variables which have been "tainted" by high inputs. For example, a variable becomes tainted if it is assigned an expression containing a high variable. If a return variable is untainted at the end of the analysis, it is considered low. Taint analyses do not consider the semantic context of the program, which leads to precision losses. For example, consider the following program, where we suppose h is high and l is low.

```
1 method f(h: Int, l: Int) returns (r: Int)
2 {
3   r := h + l
4   r := r - h
5 }
```

Our goal is to infer the NI postcondition $low(l) \Rightarrow low(r)$. A simple taint analysis considers r as tainted by h at lines 3 and 4, and cannot infer that r is low at the end of the program.

However, our framework is able to infer the desired postcondition. The results of our framework are shown below; for each abstract domain, we show the inferred specification, and we mark the strategies for which it was inferred with a cross.

Domain	Postconditions	M1	M2	M3
Oct				
Poly	`low(l) ==> low(r)`	×	×	×

To be able to infer the desired postcondition, the leaf abstract domain must be able to precisely abstract the constraints $r^{(1)} = h^{(1)} + l^{(1)}$ and $r^{(2)} = h^{(2)} + l^{(2)}$. This is achieved in the polyhedra domain, but the octagon domain is too imprecise.

When run on this example, a run of a simple static analysis (i.e., without a BDT domain) can infer that $r = h + l - h = l$ at the end of the method, from which $low(l) \Rightarrow low(r)$ can be derived. However, inferring NI specifications using such a simple static analysis is not possible in general, as shown in the next example.

BA2

This example was already shown in Figure 3.1, to illustrate the benefit of using Trace Partitioning. Our goal is to infer the postcondition $low(x) \Rightarrow low(res)$.

```
method m(x: Int) returns (res: Int)
{
  if (x >= 0) { res := 1 }
  else { res := 0 }
}
```

When run on the current example, a simple static analysis (again, without a BDT domain) can only infer the functional specification $0 \leq res \leq 1$.

On the contrary, a static analysis with a BDT domain run on the product method is capable of inferring a NI postcondition, as shown below.

Domain	Postconditions	M1	M2	M3
OCT	`low(x) ==> low(res)`	✗	✗	✗
POLY	`low(x) ==> low(res)`	✗	✗	✗

Our analysis also infers functional specifications: here, for example, we infer $0 \leq r^{(1)} \leq 1$ and $0 \leq r^{(2)} \leq 1$. In general, if a simple static analysis using an abstract domain D infers a functional specification e, our analysis (with D as leaf domain) infers the specifications $e^{(i)}$ if the i-th execution is active. For this reason, we do not show inferred functional specifications in the rest of this section, and focus on NI.

BA3

This example shows a case where the merge placement strategies M1 and M2 lead to a loss in precision, while M3 allows to infer a NI specification. Our goal is to infer the postcondition $low(x) \Rightarrow low(res)$.

```
method main(x: Int) returns (res: Int)
{
  var y: Int := 1
  if (x == 0) {
    y := 0
    res := 1
  }
  if (y != 0) { res := 2 }
}
```

Using the M1 strategy, merges on the activation variables for the if-branch (lines 5–6) are inserted in the product, and thus any information about *res* in both executions is lost after these merges.

For this example, the M2 strategy produces a result equivalent to the result of the M1 strategy. When using the M2 strategy, the set of variables occurring in the arguments of the splits for the first and second if-branch are $\{p_1^{(1)}, p_1^{(2)}, x^{(1)}, x^{(2)}\}$ and $\{p_2^{(1)}, p_2^{(2)}, y^{(1)}, y^{(2)}\}$, respectively, where $p_1^{(1)}, p_1^{(2)}$ and $p_2^{(1)}, p_2^{(2)}$ are the activation variables for the respective bodies. Since these two sets do not overlap, we merge on $p_1^{(1)}$ and $p_1^{(2)}$ after the first translated if-branch, as with the M1 strategy.

In this particular case, the heuristic used by the M2 strategy leads to a loss in precision: the partition on $p_1^{(1)}$ and $p_1^{(2)}$ can still improve precision when combined with the second splits. The M3 strategy, on the other hand, allows to infer the desired postcondition.

Domain	Postconditions	M1	M2	M3
Oct	`low(x) ==> low(res)`			×
Poly	`low(x) ==> low(res)`			×

Terauchi

This example is taken from [14], Figure 1. The desired postcondition is $low(y) \Rightarrow low(l)$.

Supposing h is high and y is low, a simple taint analysis considers x as tainted, since it is assigned conditionally on the high input h. In line 8, l becomes tainted by x, and is not considered a low output.

```
1 method main(h: Bool, y: Int) returns (l: Int)
2 {
3   var z: Int
4   var x: Int
5   z := 1
6   if (h) { x := 1 }
7   else { x := z }
8   l := x + y
9 }
```

Semantic reasoning about the values of the variables is needed to infer a NI postcondition, which is achieved by our framework.

Domain	Postconditions	M1	M2	M3
Oct				
Poly	`low(y) ==> low(l)`	×	×	×

Joana

This example (taken from [7], Figure 13, left) is similar to TERAUCHI. The desired postcondition is $low(l)$.

```
method inputPIN() returns (res: Int)

method main() returns (l: Int)
{
  var h: Int
  h := inputPIN()
  if (h < 0) { l := 0 }
  else { l := 0 }
}
```

A taint analysis considers l as tainted by h in both branches of the if-statement, and a NI postcondition cannot be inferred, even if l has the same value in both branches. Here h is considered high because `inputPIN` has no NI specification ensuring that h is low.

Our framework is able to infer the desired specification for all three merge placement strategies.

Domain	Postconditions	M1	M2	M3
OCT	`low(l)`	×	×	×
POLY	`low(l)`	×	×	×

Fibonacci

This example computes the k-th Fibonacci number, which is obviously low if k is low.

```
method fib(k: Int) returns (res: Int)
{
    var a: Int := 0
    var b: Int := 1
    var i: Int := 0
    while (i < k) {
        var tmp: Int := b
        b := a + b
        a := tmp
        i := i + 1
    }
    res := b
}
```

For this example, the desired postcondition is $low(k) \Rightarrow low(res)$, and desired loop invariants are $low(k) \Rightarrow low(i)$, $low(k) \Rightarrow low(a)$ and $low(k) \Rightarrow low(b)$.

Domain	Postconditions	M1	M2	M3
Oct				
Poly	`low(k) ==> low(res)`	✗	✗	✗

Domain	Invariants	M1	M2	M3
Oct	`low(k) ==> low(i)`	✗	✗	✗
Poly	`low(k) ==> low(i)`	✗	✗	✗
	`low(k) ==> low(a)`	✗	✗	✗
	`low(k) ==> low(b)`	✗	✗	✗

Küsters

This example (taken from [9]) is a Viper encoding of a Java program. Both *a* and *result* are static variables in the Java program, which we encode by explicitly passing a reference parameter *global*. The body of `bar` is not specified.

```
field result: Int
field a: Int

method main(global: Ref, secret: Int)
  requires acc(global.a, 1)
  requires acc(global.result, 1)
  ensures acc(global.a, 1)
  ensures acc(global.result, 1)
{
  global.a := 42
  bar(secret)
  var b: Int
  b := foo(global, secret)
  global.result := b
}

method foo(global: Ref, secret: Int) returns (res: Int)
  requires acc(global.a, 1/2)
  ensures acc(global.a, 1/2)
{
  var b: Int
  b := global.a
  if (secret == 0) { b := b + secret }
  res := b
}

method bar(secret: Int) // Hidden implementation
```

To ensure soundness, our analysis needs to know which heap locations are modified in the body of called methods. In Viper, this information can be encoded using *access permissions*. `acc(`$r.f$`, 1)` represents a write permission on the heap location $r.f$, and `acc(`$r.f$`, `p`)` for $p < 1$ represents a read permission.

To get access predicates for every method, a pre-analysis could be run on the program in order to infer access permissions. Since the inference of access permissions is largely orthogonal to the topic of this thesis, we assume that methods have been annotated with correct access permissions by the user.

Since `main` calls `foo` in its body, an interprocedural analysis is needed to infer NI specifications for `main`.

For non-recursive method calls, an interprocedural analysis can be simulated by inferring specifications for the callee (using an intraprocedural analysis), and using them to transform the abstract state at the method call site in the caller. In general, we would build the call graph of the program (in which each node represents a method, and there is a directed edge from node `m` to node `n` if the method `m` calls the method `n`).

Without recursive calls, this graph is loop-free, and a topological sorting of the graph can be computed. Since the product program allows to reason modularly about hyperproperties, we could infer specifications for the last method in the topological order, and use these specifications to infer specifications for the second to last method, and so on up until the root method.

The implementation of such an approach is beyond the scope of this thesis, we simulate the described process manually.

Our analysis produces the following postcondition for the method `foo`:

Domain	Postconditions	M1	M2	M3
H×Oct	`low(old(global.a)) ==> low(res)`	×	×	×
H×Poly	`low(old(global.a)) ==> low(res)`	×	×	×

In Viper, the `old` keyword (used in postconditions) allows one to reference the value of an expression at the beginning of a method. Sample does not support `old` expressions: we therefore transform `foo`'s postcondition $low(old(global.a)) \Rightarrow low(res)$ into a precondition $low(global.a)$ and a postcondition $low(res)$. This pair of pre- and postcondition is then used to infer the following postconditions for `main`:

Domain	Postconditions	M1	M2	M3
H×Oct	`low(global.a)`	×	×	×
	`low(global.result)`	×	×	×
H×Poly	`low(global.a)`	×	×	×
	`low(global.result)`	×	×	×

ZeroEntries

The method `zero_entries` below counts the number of zero-entries in the Viper sequence `A`.

```
method zero_entries(A: Seq[Int]) returns (count: Int)
{
  count := 0
  var i: Int := 0
  while (i < |A|) {
    if (A[i] == 0) { count := count + 1 }
    i := i + 1
  }
}
```

Our framework does not support Viper sequences in their native form, as they appear in this program. Therefore, we encode this method such that it uses references, as shown below. Since our framework provides a leaf domain capable of abstracting heap locations, we can infer interesting specifications on this new method.

The length of the sequence is encoded as an `Int` field. An array access `A[i]` is encoded as a method call `get(A, i)`. The method `get` guarantees, via its postcondition, that `get(A, i)` is low if both `A` and `i` are low.

```
field length: Int

method get(A: Ref, i: Int) returns (res: Int)
  requires acc(A.length, 1/2)
  requires 0 <= i && i < A.length
  ensures (low(A) && low(i)) ==> low(res)
  ensures acc(A.length, 1/2)

method zero_entries(A: Ref) returns (count: Int)
  requires acc(A.length, 1/2)
  ensures acc(A.length, 1/2)
{
  count := 0
  var i: Int := 0
  while (i < A.length) {
    var x: Int
    x := get(A, i)
    if (x > 0) { count := count + 1 }
    i := i + 1
  }
}
```

In the product program, we add two assumptions to ensure that properties that hold for sequence still hold in the rewritten method. These assumptions are:

- $(p^{(1)} \Rightarrow A^{(1)} \neq null) \wedge (p^{(2)} \Rightarrow A^{(2)} \neq null)$, since a Viper sequence is never null.
- $(p^{(1)} \wedge p^{(2)}) \Rightarrow A^{(1)} = A^{(2)} \Rightarrow A^{(1)}.length^{(1)} = A^{(2)}.length^{(2)}$, i.e., if a `A` is low, then `A.length` is low as well. To ensure that the method

annotated by our inferred specifications can be directly fed to a verifier, we translate our assumptions and add them as preconditions to the original program. Thus, the resulting method will have preconditions `A != null` and `low(A) ==> low(A.length)`.

Applying our analysis to the rewritten method, we can infer the following postconditions and loop invariants:

Domain	Postconditions	M1	M2	M3
H×Oct	`low(A) ==> low(count)`	×	×	×
H×Poly	`low(A) ==> low(count)`	×	×	×

Domain	Invariants	M1	M2	M3
H×Oct	`low(A) ==> low(i)`	×	×	×
	`low(A) ==> low(count)`	×	×	×
H×Poly	`low(A) ==> low(i)`	×	×	×
	`low(A) ==> low(count)`	×	×	×

We believe that the translation from a program using sequences to an equivalent program using the heap can be fully automated. Expressions `|A|` are translated as `A.length`. For a statement containing an access `A[i]`, we assign `get(A, i)` to a fresh temporary variable, which replaces `A[i]` in the translated statement. The assumptions described above hold for all sequences, and can thus be inserted in the translated program automatically. However, specifications containing accesses `A[i]` cannot be translated to a method call `get(A, i)`, since only pure expressions are allowed in Viper specifications.

Eilers

This example (adapted from [6], Figure 1) is similar to the example ZeroEntries presented above.

```
method is_female(person: Int) returns (res: Int)
{
  if (person >= 0) { res := 1 } else { res := 0 }
}

method main(people: Seq[Int]) returns (count: Int)
  requires acc(people.length, 1/2)
  ensures acc(people.length, 1/2)
{
  var i: Int := 0
  count := 0
  while (i < |people|)
  {
    var current: Int
    current := people[i]
    var female : Int
    female := is_female(current)
    count := count + female
    i := i + 1
  }
}
```

As before, we rewrite the method to use objects, and make the same assumptions. The rewritten method is shown below.

```
field length: Int
method get(arr: Ref, i: Int) returns (res: Int)
  requires acc(arr.length, 1/2)
  requires 0 <= i && i < arr.length
  ensures (low(arr) && low(i)) ==> low(res)
  ensures acc(arr.length, 1/2)

method is_female(person: Int) returns (res: Int)
{
  if (person >= 0) { res := 1 } else { res := 0 }
}

method main(people: Ref) returns (count: Int)
  requires acc(people.length, 1/2)
  ensures acc(people.length, 1/2)
{
  var i: Int := 0
  count := 0
  while (i < people.length)
  {
    var current: Int
    current := get(people, i)
    var female : Int
    female := is_female(current)
    count := count + female
    i := i + 1
  }
}
```

Simulating an interprocedural analysis manually, as for Küsters, we first infer the following postcondition for `is_female`:

Domain	Postconditions	M1	M2	M3
H×Oct	`low(person) ==> low(res)`	×	×	×
H×Poly	`low(person) ==> low(res)`	×	×	×

This postcondition is then used to infer the following postconditions and invariants for `main`:

Domain	Postconditions	M1	M2	M3
H×Oct				
H×Poly	`low(people) ==> low(count)`	×	×	×

Domain	Invariants	M1	M2	M3
H×Oct	`low(people) ==> low(i)`	×	×	×
H×Poly	`low(people) ==> low(i)`	×	×	×
	`low(people) ==> low(count)`	×	×	×

5.1.2 Performance

Program	Loop	Call	Heap	Domain	M1	M2	M3
BA1				Oct	0.01*	0.01*	0.01*
				Poly	0.04	0.05	0.02
BA2				Oct	0.04	0.04	0.06
				Poly	0.11	0.12	0.17
BA3				Oct	0.06*	0.04*	0.08
				Poly	0.29*	0.11*	0.19
Terauchi				Oct	0.09*	0.06*	0.08*
				Poly	0.41	0.40	0.48
Joana		×		Oct	0.03	0.02	0.03
				Poly	0.10	0.05	0.07
Fibonacci	×			Oct	0.44*	0.42*	0.42*
				Poly	2.05	2.10	2.03
Küsters		×	×	H×Oct	0.54	0.44	1.39
				H×Poly	2.69	3.04	7.03
ZeroEntries	×		×	H×Oct	3.34	3.23	3.37
				H×Poly	10.2	14.1	13.5
Eilers	×	×	×	H×Oct	7.17*	6.87*	6.95*
				H×Poly	10.2	10.4	9.56

Table 5.1: Evaluated examples. We show the used language features and the leaf domain. The rightmost column shows the running time (in seconds) of the analysis for the merge placement strategies M1, M2, and M3. Entries marked with an asterisk denote combinations of parameters for which no meaningful specification could be inferred.

The running times of the analysis for the examples presented in this section are shown in Table 5.1.

For all but one example from this corpus, the desired specifications can be inferred using the M1 strategy. For the remaining example BA3, the M3 strategy allows us to attain the necessary precision.

Using Oct or H×Oct as our leaf domain, we can infer NI specifications for 5 out of 9 examples; when using Poly or H×Poly, we can infer specifications for all of them, including those containing loops, method calls and heap-manipulating statements. For programs containing loops, we are able to infer NI loop invariants.

All specifications inferred using Oct and Poly can also be inferred using H×Oct and H×Poly, respectively, albeit slightly slower (because the analysis carries an additional top heap abstract state).

The complexity of the leaf domain has a strong influence on the runtime. A reduced product of a heap domain with a numerical domain stores more information than the numerical domain alone, and is thus typically slower. Similarly, the polyhedra domain is more precise than the octagon domain, and typically leads to a slower analysis. In addition, the Java bindings for Apron (the library implementing the polyhedra abstract domain) are not thread-safe, which prevented us from using parallelism for Poly and H×Poly leaf domains. Moreover, our implementation of the storeless heap domain is not optimized, which certainly penalized the efficiency of the analysis for heap-manipulating programs.

On this corpus, the average increase in runtime when using Poly instead of Oct (or H×Poly instead of H×Oct) is 273% for M1, 319% for M2, and 253% for M3.

In general, if a program contains an assignment of the form $x := y + z$, a polyhedra leaf domain is necessary to infer NI specifications, because the octagon domain does not store the constraint $x = y + z$ precisely.

5.2 Comparator Implementations

In this section, our goal is to infer Antisymmetry specifications. The examples in this section are Java examples taken from [13], which we encoded in Viper. The Viper code of these examples is shown in Appendix A.

Each example contains a method `compare(a: Ref, b: Ref)` with an `Int` return variable `r`, which implements the Java Comparator interface. Our goal is to infer the postcondition $p^{(1)} \wedge p^{(2)} \Rightarrow a^{(1)} = b^{(2)} \wedge a^{(2)} = b^{(1)} \Rightarrow r^{(1)} = -r^{(2)}$ on the products, corresponding to the antisymmetry property $\texttt{compare}(a,\ b) = -\texttt{compare}(b,\ a)$.

In order to achieve this, we insert the assumption $a^{(1)} = b^{(2)} \wedge a^{(2)} = b^{(1)}$ at the start of the `compare` methods.

For Symmetry-like properties (i.e., properties we can infer by swapping inputs in the executions), we need not duplicate the fields in the product, as described in Section 2.2.2. Instead, the original fields could be kept in the product, and a field access $o.f$ could then be translated as $o^{(i)}.f$ in the i-th execution.

For the sake of generality, we did not use this simpler encoding of the heap for the examples in this section. Thus, it is necessary to add an assumption split with condition $o.f^{(1)} = o.f^{(2)}$ for each reference parameter o and each field f in the product.

An element of the (storeless) heap domain abstracting the constraint $a^{(1)} = b^{(2)} \wedge a^{(2)} = b^{(1)}$ for reference variables $a^{(1)}, a^{(2)}, b^{(1)}, b^{(2)}$ maps $a^{(1)}$, $b^{(2)}$ to $\{O_1\}$, and $a^{(2)}$, $b^{(1)}$ to $\{O_1, O_2\}$ for some abstract objects O_1, O_2. However, this abstraction is too imprecise to say with certainty that $a^{(2)} = b^{(1)}$. To palliate this imprecision, we insert a split on $a^{(1)} = a^{(2)}$, which separates the cases where $a^{(1)} = b^{(2)} = a^{(2)} = b^{(1)}$ (in the left subtree) and $a^{(1)} = b^{(2)} \neq a^{(2)} = b^{(1)}$ (in the right subtree). This split has the additional benefit of considering the case `compare(`a`, `a`)` explicitly. This additional split is not tailored to our examples or to the desired specification, and can be used for general Symmetry-like hyperproperties for heap-manipulating methods.

Using our approach, we are able to infer the desired postcondition for all shown examples. Additionally, we infer the postcondition $p^{(1)} \wedge p^{(2)} \Rightarrow a^{(1)} = b^{(2)} \wedge a^{(2)} = b^{(1)} \wedge a^{(1)} = a^{(2)} \Rightarrow r^{(1)} = 0$ (and similarly for $r^{(2)}$) for all shown examples. This postcondition corresponds to the reflexive property `compare(`a`, `a`)`$= 0$.

We show the running times in Table 5.2. For each example, we ran the analysis using H×Oct and H×Poly, and measured the runtime for the simplest domain with which we could infer the desired specification.

Program	Fields	Domain	M1	M2	M3
Time	2	H×Oct	3.71	3.73	3.78
Container	4	H×Oct	10.0	14.5	12.5
Match	3	H×Poly	2.53	2.72	3.75
CollItem	4	H×Poly	11.8	12.1	19.3

Table 5.2: Running times (in seconds) for implementations of the Java Comparator interface, translated to Viper.

Not all examples from [13] are shown here. Some of the other examples contain language constructs that our framework does not handle, e.g. `Rational` variables. The remaining examples contain objects which behave as collections, i.e., which contain a method `get(`i`)` returning the i-th element. Due to the construction of the product, the pure function `get` is not encoded with enough precision in the product to allow the inference of Symmetry-like specifications. We encode the pure function `get(i: Int)` as a Viper method `get(r: Ref, i: Int)` annotated with a postcondition ensuring determinism. In the product, this guarantees that after executing the statement

$$x^{(1)}, x^{(2)} := \texttt{get}(p^{(1)},\ p^{(2)},\ r^{(1)},\ r^{(2)},\ i^{(1)},\ i^{(2)}),$$

$x^{(1)} = x^{(2)}$ if $r^{(1)} = r^{(2)}$ and $i^{(1)} = i^{(2)}$, if both activation variables are true. However, due to the construction of the product, we cannot specify that after executing

$$\begin{aligned} x^{(1)}, x^{(2)} &:= \texttt{get}(p^{(1)},\ p^{(2)},\ a^{(1)},\ a^{(2)},\ i^{(1)},\ i^{(2)}); \\ y^{(1)}, y^{(2)} &:= \texttt{get}(p^{(1)},\ p^{(2)},\ b^{(1)},\ b^{(2)},\ i^{(1)},\ i^{(2)}), \end{aligned}$$

$x^{(1)} = y^{(2)}$ if $a^{(1)} = b^{(2)}$ and $i^{(1)} = i^{(2)}$, provided both executions are active. This causes our framework to be imprecise for such cases. This loss in precision stems from the construction of the program itself. It is also worth noting that this problem occurs when equalities between different variables in different executions appear in the left hand side of the hyperproperty. For example, NI hyperproperties do not suffer from this problem, because only equalities between the same variable in different executions appear on the left hand side.

5.3 Miscellaneous Examples

In this section, we infer Symmetry, Reflexivity and Monotonicity specifications.

5.3.1 Euclidean Algorithm - Symmetry and Reflexivity

A Viper implementation of Euclid's algorithm for the computation of the greatest common divisor of two integers is shown below.

```
method gcd(i: Int, j: Int) returns (res: Int)
{
    var a: Int := i
    var b: Int := j
    while (a != b) {
        if (a > b) { a := a - b }
        else { b := b - a }
    }
    res := a
}
```

For this example, our goal is to infer a Symmetry-like specification about `gcd`, i.e., a relationship between `gcd(`i`,` j`)` and `gcd(`j`,` i`)`.

Similarly to the examples in Section 5.2, we insert the assumption $i^{(1)} = j^{(2)} \land i^{(2)} = j^{(1)}$, which states that the inputs are swapped in the two executions. As before, we also insert a split on $i^{(1)} = i^{(2)}$. In this case, this split is not necessary (it only increases the precision of a heap leaf domain), but it has the advantage of making the case `gcd(`i`,` i`)` explicit. Generally, we insert this split for all Symmetry-like hyperproperties, since we can then infer Reflexivity specifications along with Symmetry-like specifications.

The postconditions inferred by our framework are shown below. For the sake of readability, we write A for `rel(i, 1) == rel(j, 2) && rel(i, 2) == rel(j, 1)` and A' for A `&& rel(i, 1) == rel(i, 2)`.

Domain	Postconditions	M1	M2	M3
Oct	A' `==> rel(res, 1) == rel(res, 2)`	×	×	×
Poly	A `==> rel(res, 1) == rel(res, 2)`	×	×	×
	A' `==> rel(res, 1) == rel(i, 1)`	×	×	×
	A' `==> rel(res, 2) == rel(i, 2)`	×	×	×

Thus, when using Poly, we can infer that `gcd(`i`,` j`)` $=$ `gcd(`j`,` i`)`, and that `gcd(`i`,` i`)` $= i$.

Our framework also infers the following loop invariants:

Domain	Invariants	M1	M2	M3
Oct				
Poly	A `==> rel(a, 1) == rel(b, 2)`	×	×	×
	A `==> rel(a, 2) == rel(b, 1)`	×	×	×
	A' `==> rel(a, 1) == rel(i, 1)`	×	×	×
	A' `==> rel(a, 2) == rel(i, 2)`	×	×	×

The first two loop invariants state that if i and j are swapped in the two executions, then a and b are also swapped. The last two loop invariants state that if $i = j$, $a = i$ holds in both executions.

The (averaged) running times are below 0.5 seconds for all three strategies, for both Oct and Poly.

5.3.2 Monotonicity of Multiplication

The following Viper method returns the multiplication of its two integer arguments (provided they are non-negative).

```
method mult(x: Int, y: Int) returns (res: Int)
{
  var i: Int
  i := 0
  res := 0
  while (i < y) {
    res := res + x
    i := i + 1
  }
}
```

Our goal is to infer properties about the monotonicity of multiplication, i.e., $x \leq x' \Rightarrow x \cdot y \leq x' \cdot y$ for non-negative x, x', and y.

The translation of this specification in the product is $p^{(1)} \wedge p^{(2)} \Rightarrow x^{(1)} \leq x^{(2)} \wedge y^{(1)} = y^{(2)} \Rightarrow res^{(1)} \leq res^{(2)}$.

To infer the desired specification, we insert an initial split on the condition $x^{(1)} \leq x^{(2)} \wedge y^{(1)} = y^{(2)}$, which appears in the left hand side of our specification. This split is not tailored to our example, and can be inserted for all programs on which we want to infer Monotonicity specifications.

The postconditions inferred by our framework are shown below. We write A for `rel(x, 1) <= rel(x, 2) && rel(y, 1) == rel(y, 2)`.

Domain	Postconditions	M1	M2	M3
Oct				
Poly	A `==> rel(res, 1) <= rel(res, 2)`	×	×	×

We also infer the following loop invariants:

Domain	Invariants	M1	M2	M3
Oct	A `==> rel(i, 1) == rel(i, 2)`	×	×	×
Poly	A `==> rel(i, 1) == rel(i, 2)`	×	×	×
	A `==> rel(res, 1) <= rel(res, 2)`	×	×	×

For all three merge strategies, the averaged running times were below 0.4 seconds for Oct, and below 1.3 seconds for Poly.

5.4 Discussion

Our methodology allows to infer various hyperproperties (NI, Symmetry, Monotonicity) on programs, without having to develop an analysis specific to a given hyperproperty. Our approach can infer NI specifications where conventional approaches (e.g., taint analyses) are too imprecise to yield useful results.

Our approach handles non-trivial language features like loops and heap-manipulating programs. Additionally, we can encode sequences as references, infer specifications for the encoded programs, which can then be translated back to the original.

We can simulate an interprocedural analysis for non-recursive methods by inferring specifications on the callee, and applying them at the call site in the caller.

We have evaluated our framework on a corpus of examples using both the octagon and polyhedra domains in the leaves of our BDT domain. The polyhedra domain is more precise than the octagon domain and allows to infer more specifications in general, at the expense of speed.

The precision of our abstract domain is limited by the precision of the leaf domain. As an example, a method `R(`a`:Int, `b`:Int)` with a boolean return variable *res* is said to be total if `R(`a`, `b`)` $\lor$ `R(`b`, `a`)` for all a and b. Totality is a 2-hyperproperty, and can be expressed as $p^{(1)} \land p^{(2)} \Rightarrow a^{(1)} = b^{(2)} \land a^{(2)} = b^{(1)} \Rightarrow \mathit{res}^{(1)} \lor \mathit{res}^{(2)}$ in a 2-product. To be able to infer such a specification, a leaf abstract domain with disjunctions (i.e., capable of expressing the constraint $\mathit{res}^{(1)} \lor \mathit{res}^{(2)}$ precisely) is needed.

Similarly, `R` implements a transitive binary relation on integers if `R(`a`, `b`)` $\land$ `R(`b`, `c`)` $\Rightarrow$ `R(`a`, `c`)`. This 3-hyperproperty can be expressed in a 3-product as $p^{(1)} \land p^{(2)} \land p^{(3)} \Rightarrow a^{(1)} = a^{(3)} \land b^{(1)} \land a^{(2)} \land b^{(2)} = b^{(3)} \Rightarrow \neg \mathit{res}^{(1)} \lor \neg \mathit{res}^{(2)} \lor \mathit{res}^{(3)}$. Again, this specification requires a disjunctive leaf domain to be inferred.

The implementation of such a disjunctive domain is beyond the scope of this thesis, which is why this chapter contains no examples for totality and transitivity. Nevertheless, we believe that given an implementation of a disjunctive abstract domain, such hyperproperties can be inferred without much additional effort.

Chapter 6

Conclusion and Future Work

In this thesis, we developed a static analysis for the inference of general hyperproperty specifications.

Our analysis constructs a product program, which simulates several executions of the original program, and use Abstract Interpretation to infer specifications on the product. To compensate the loss in program structure introduced by the product construction, we used trace partitioning to restore the control flow information in the product. This is achieved by instrumenting the product program with commands, which inform the analysis of the partitions on the traces it has to execute. To keep the size of our abstract domain reasonably small, we also inserted commands which explicitly inform the analysis of partitions it should forget about. This instrumentation of the product program is done statically, before running the analysis.

We also insert commands in the product program which tell the analysis to partition traces on interesting properties of inputs. For a given hyperproperty, these partitioning expressions are conditions on inputs appearing on the left-hand-side of an implication.

Our abstract domain uses Binary Decision Trees to store partitioning information, and is parametric in its leaf abstract domain. This allows users to provide abstract domains that are relevant to their specific applications, and lets them tune the precision-speed trade-off of the domain.

Using our approach, we infer Non-Interference, Symmetry and Monotonicity properties on programs, which include loops, method calls, heap manipulations, and sequences.

The main drawback of our methodology is the size of our abstract states, which grows exponentially with the number of trace partitions. Various optimizations have been implemented to speed up the analysis, but more work can certainly be done to speed it up even more.

A possible extension to this thesis could explore other hyperproperties not discussed or evaluated in this report. Further implementation efforts could also be considered, e.g., a backward analysis to infer preconditions of methods, or a truly interprocedural analysis capable of handling recursive methods.

Appendix A

Appendix

The Viper examples evaluated in Section 5.2 are shown below.

Time

```
field ora: Int
field volume_totale : Int

method Int_compare(o1: Int, o2: Int) returns (res: Int)
  ensures (rel(o1, 0) == rel(o2, 1) && rel(o2, 0) == rel(o1, 1))
          ==> rel(res, 0) == -rel(res, 1)

method compare(o1: Ref, o2: Ref) returns (res: Int)
  requires acc(o1.ora, 1/2)
  requires acc(o2.ora, 1/2)
  requires acc(o1.volume_totale, 1/2)
  requires acc(o2.volume_totale, 1/2)
  ensures acc(o1.ora, 1/2)
  ensures acc(o2.ora, 1/2)
  ensures acc(o1.volume_totale, 1/2)
  ensures acc(o2.volume_totale, 1/2)
{
  var cmp : Int
  cmp := Int_compare(o1.ora, o2.ora)
  if (cmp == 0){
    cmp := Int_compare(o1.volume_totale, o2.volume_totale)
  }
  res := cmp
}
```

Container

```
field departureTime: Int
field departureMaxDuration: Int
field departureTransportCompany: Int
field departureTransportType: Int

method Int_compare(o1: Int, o2: Int) returns (res: Int)
  ensures (rel(o1, 0) == rel(o2, 1) && rel(o2, 0) == rel(o1, 1))
          ==> rel(res, 0) == -rel(res, 1)

method compare(o1: Ref, o2: Ref) returns (res: Int)
  requires acc(o1.departureTime, 1/2)
  requires acc(o2.departureTime, 1/2)
  requires acc(o1.departureMaxDuration, 1/2)
  requires acc(o2.departureMaxDuration, 1/2)
  requires acc(o1.departureTransportCompany, 1/2)
  requires acc(o2.departureTransportCompany, 1/2)
  requires acc(o1.departureTransportType, 1/2)
  requires acc(o2.departureTransportType, 1/2)

  ensures acc(o1.departureTime, 1/2)
  ensures acc(o2.departureTime, 1/2)
  ensures acc(o1.departureMaxDuration, 1/2)
  ensures acc(o2.departureMaxDuration, 1/2)
  ensures acc(o1.departureTransportCompany, 1/2)
  ensures acc(o2.departureTransportCompany, 1/2)
  ensures acc(o1.departureTransportType, 1/2)
  ensures acc(o2.departureTransportType, 1/2)
{
  var rv : Int

  rv := Int_compare(o1.departureTime, o2.departureTime)
    if (rv == 0) {
      rv := Int_compare(o1.departureMaxDuration,
                        o2.departureMaxDuration)
      if (rv == 0) {
        rv := Int_compare(o1.departureTransportCompany,
                          o2.departureTransportCompany)
        if (rv == 0) {
          rv := Int_compare(o1.departureTransportType,
                            o2.departureTransportType)
        }
      }
    }
  res := rv
}
```

Match

```
field score: Int
field seq1start: Int
field seq2start: Int

method Int_compare(o1: Int, o2: Int) returns (res: Int)
  ensures (rel(o1, 0) == rel(o2, 1) && rel(o2, 0) == rel(o1, 1))
          ==> rel(res, 0) == -rel(res, 1)

method compare(o1: Ref, o2: Ref) returns (res: Int)
  requires acc(o1.score, 1/2)
  requires acc(o2.score, 1/2)
  requires acc(o1.seq1start, 1/2)
  requires acc(o2.seq1start, 1/2)
  requires acc(o1.seq2start, 1/2)
  requires acc(o2.seq2start, 1/2)

  ensures acc(o1.score, 1/2)
  ensures acc(o2.score, 1/2)
  ensures acc(o1.seq1start, 1/2)
  ensures acc(o2.seq1start, 1/2)
  ensures acc(o1.seq2start, 1/2)
  ensures acc(o2.seq2start, 1/2)
{
  res := Int_compare(o1.score, o2.score)
  if (res == 0) {
    res := Int_compare(o1.seq1start + o1.seq2start,
                       o2.seq1start + o2.seq2start)
  }
}
```

CollItem

```
field getCardSet: Int
field getCardRarity: Int
field getCardId: Int
field cardType: Int

method Int_compare(o1: Int, o2: Int) returns (res: Int)
  ensures (rel(o1, 0) == rel(o2, 1) && rel(o2, 0) == rel(o1, 1))
          ==> rel(res, 0) == -rel(res, 1)


method compare(o1: Ref, o2: Ref) returns (res: Int)
  requires acc(o1.getCardSet, 1/2)
  requires acc(o2.getCardSet, 1/2)
  requires acc(o1.getCardRarity, 1/2)
  requires acc(o2.getCardRarity, 1/2)
  requires acc(o1.getCardId, 1/2)
  requires acc(o2.getCardId, 1/2)
  requires acc(o1.cardType, 1/2)
  requires acc(o2.cardType, 1/2)

  ensures acc(o1.getCardSet, 1/2)
  ensures acc(o2.getCardSet, 1/2)
  ensures acc(o1.getCardRarity, 1/2)
  ensures acc(o2.getCardRarity, 1/2)
  ensures acc(o1.getCardId, 1/2)
  ensures acc(o2.getCardId, 1/2)
  ensures acc(o1.cardType, 1/2)
  ensures acc(o2.cardType, 1/2)
{
  if (o1 == o2){
    res := 0
  } else {
    res := Int_compare(o1.getCardSet, o2.getCardSet)
    if (res == 0) {
      res := Int_compare(o1.getCardRarity, o2.getCardRarity)
      if (res == 0) {
        res := Int_compare(o1.getCardId, o2.getCardId)
        if (res == 0) {
          res := o1.cardType - o2.cardType
        }
      }
    }
  }
}
```

Bibliography

[1] J. Chen and P. Cousot. A binary decision tree abstract domain functor. In S. Blazy and T. Jensen, editors, *Static Analysis - 22nd International Symposium (SAS 2015)*, volume 9291 of *LNCS*, pages 36–53. Springer, 2015.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[3] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[4] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In M. Hofmann, editor, *14th International Conference on Fondations of Software Science and Computation Structures (FoSSaCS 2011), Saarbrücken, Germany*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer-Verlag, Heidelberg, March 26 – April 3, 2011.

[5] Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 665–687, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] M. Eilers, P. Müller, and S. Hitz. Modular product programs. In *European Symposium on Programming (ESOP)*, 2018.

[7] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, Jun 2015.

[8] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 661–667, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[9] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A hybrid approach for proving non-interference of java programs. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 305–319. IEEE Computer Society, July 2015.

[10] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP)*, volume 3444 of *LNCS*. Springer, 2005.

[11] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006. http://www-apr.lip6.fr/~mine/publi/article-mine-HOSC06.pdf.

[12] 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.

[13] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, New York, NY, USA, 2016. ACM.

[14] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 352–367, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.