Different Maps for Different Uses
A Program Transformation for Intermediate Verification Languages

Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski

University of Freiburg

Abstract. In theorem prover or SMT solver based verification, the program to be verified is often given in an intermediate verification language such as Boogie, Why, or CHC. This setting raises new challenges. We investigate a preprocessing step which takes the similar role that alias analysis plays in verification, except that now, a (mathematical) map is used to model the memory or a data object of type `array`. We present a program transformation that takes a program P to an equivalent program P' such that, by verifying P' instead of P, we can reduce the burden of the exponential explosion in the number of case splits. Here, the case splits are according to whether two statements using the same map variable are independent or not; if they are independent, we might as well employ two different map variables and thus remove the need for a case split (this is the idea behind the program transformation). We have implemented the program transformation and show that, in an ideal case, we can avoid the exponential explosion.

1 Introduction

In theorem prover or SMT solver based verification, the program to be verified is often given in an *intermediate verification language* (such as Boogie [10], Why [6], or CHC [7]). This setting is useful in many aspects but it raises its proper challenges; see, e.g., [2] for the investigation of axiomatic semantics. Here, we investigate a novel problem that arises in this setting where a (mathematical) map is used to model the memory or a data object of type `array`. The problem is to transform a program into an equivalent program such that statements with independent uses of a given map variable become statements with different map variables. In a way, we lift the *alias problem* from programming languages to intermediate verification languages. We will next explain the problem and the new challenge that it raises. The explanation is subtle and will need a large chunk of the introduction.

The idea behind an intermediate verification language is the one of a *lingua franca* for verification. Once a C or Java program has been translated to intermediate code, we are no longer bothered with the intricacies and ambiguities of definitions of programming language semantics. There are no hidden assumptions (such as, e.g., the absence of undefined behavior); the program is taken *as is*, i.e., all assumptions appear in the program text (e.g., in `ensure` statements). This is one reason why it has been advocated to present benchmarks in an intermediate programming language for software verification competitions; see, e.g., [1]. Note that there are several scenarios where the program in the intermediate verification language comes without a corresponding program in a programming language. For example, it may have been constructed by a specific module of the verification method; see, e.g., the construction of *path programs* in [3] and [8].

The data manipulated by a program in an intermediate verification language are mathematical objects (in the same domains and logical theories that underly the theorem prover or SMT solver used for the verification). In particular, an object of type `array` in the intermediate verification language is, in fact, a map in the mathematical sense (i.e., it is manipulated like a mathematical map).

The importance of maps in intermediate verification languages is inherited directly from the importance of arrays in programming languages. The importance is amplified by the fact that in verification it is often convenient to view the memory (or, the *heap*) as a special case of an array.

What is also inherited is, unfortunately, a notorious practical issue in program verification: the need of case splits according to whether two statements with a write resp. read access to a given array (or, to the memory) refer to the same position, or not. A well-known consequence of such case splits is that they can lead to the exponential explosion of the size of the verification condition. If before we had the exponential explosion in the number of statements in the program that access a given array (or the memory), we have now have the exponential explosion in the number of statements that use a given map. We thus need to address an analogous issue in the context of intermediate verification languages.

The standard solution to address the notorious practical issue is a preprocessing step with an *alias analysis.* Roughly speaking, the alias analysis can help to infer which case splits are redundant. In some cases, the alias analysis can thus alleviate the burden of the exponential explosion in the number of case splits.

Unfortunately, an alias analysis for programming languages cannot readily be transferred to a solution for intermediate verification languages. The new challenge stems from the fact that we assume that a program in the intermediate verification language will encode every assumption in the program text; i.e., we are not allowed to use any assumption that does not appear in the program text.

We give an example to illustrate this point. The example program is depicted in Figure 3 in Section 10. We here use the map-valued variable `mem` to model the memory and the procedure `malloc` to model allocation (which we can specify together with `ensure` statements that encode our assumptions about allocation). A statement that uses the map `mem` at position `p` intuitively models the access of memory (by a write or by a read) via the pointer variable `p`. We take a program that contains two statements which use `mem` at position `p` and position `q`, respectively. We would like to infer that the uses of `mem` in the two statements are independent. The term *independent* here means that the value of `p` in the execution of the one statement is different from the value of `q` in the execution of the other statement (in every execution of the program). In our setting, we are not allowed to use any hidden assumption (such as the absence of undefined behavior). For example, we are not allowed to assume that there is no execution in which the two statements are executed when the value of `p` and `q` is `null`. Thus, we are not allowed to conclude that the two uses of `mem` in the two statements are independent even if we can infer that `p` has not been assigned to `q`, and vice versa. This would not be *sound*.

Note that in the context of programming languages, where it is common to use the assumption of the absence of undefined behavior, it would be considered sound to conclude "`p` and `q` do not alias" if the analysis can infer from the property that there is no execution in the program that assigns `p` to `q`, and vice versa. In this sense, the hidden assumption is the basis for the existence of very efficient (and effective) alias analyses. A static analysis can infer the property by checking a strong sufficient condition for the property (e.g., that the corresponding statements simply do not occur in the program).

Contributions The overall contribution of this paper is to investigate the theoretical foundations and a preliminary solution for a novel research question which may be relevant for the practical potential of intermediate verification languages.

The question concerns a preprocessing step for intermediate verification languages which takes the similar role that alias analysis plays in the verification for programming languages. Since it is convenient to implement an optimization as a program transformation (in particular for intermediate code), we consider a program transformation that takes a program P to an equivalent program P' such that, by verifying P' instead of P, we can reduce the burden of the exponential explosion in the number of case splits. Here, the case splits are according to whether two statements in P using the same map variable are independent or not; if they are independent, we might as well employ different map variables and thus remove the need for a case split (this is the idea behind the program transformation). The question is: Does there exist such a program transformation, and can it be made scalable?

In this paper, we present such a program transformation, together with its implementation which we use to show that, in the best case, we can avoid the exponential explosion altogether.

The program transformation is based on a static analysis that conservatively infers which statements using a giving map variable are independent. The overall goal of the analysis is to infer a *grouping* of statements such that we can introduce a different map variable for each group of statements (the statements within each group use the same map variable).

Our technical contributions are as follows.

– We formally introduce the independence property which enables the desired program transformation (in the context for of the intermediate programming language).

– We present a static analysis that conservatively infers which statements using a giving map variable are independent. We define an instrumentation of a program with auxiliary variables such that an existing static analysis can infer the independence property.
– We define a program transformation that takes as input a program and the inferred independence property and returns a new program. In the new program statements use different map variables according to the inferred indepence property.
– We prove that the program transformation is sound, i.e., the new program is bisimulation equivalent to the input program.
– We have implemented the program transformation into a toolchain for automatic verification. A preliminary experimentation shows that the program transformation can be effective, at least in principle. On a benchmark suite which is specifically tailored to condensate the case split explosion problem, the toolchain with the program transformation scales very well in the size of the program (whereas the toolchain without the program transformation quickly falls into the case split explosion problem and runs out of time or space).

2 Example

The left hand side of Figure 1 shows an example program given in the Boogie [10] verification language. While the program models a program in the C programming language we want to stress that our technique cannot rely on any metainformation specific to C, like the meaning of the `malloc` procedure, or the absence map reads on uninitialized cells. Map semantics in Boogie follow McCarthy's theory of arrays [11], which is also used in SMT solvers.

The example program is artificial. Its purpose is to necessitate a large number of non-interference checks in a program of minimal size. So the main obstacle to verifcation is the necessity of proving non-interference between the map updates.

In the example, dynamically allocated memory is modeled by the two map variables `mem` and `valid`. The map `mem` stores the contents of the memory. The map `valid` stores which memory cells are allocated. C's malloc function is modeled by the procedure `malloc`, which returns a memory location that is not currently in use. (For simplicity we assume that all memory blocks are of size 1.)

The procedure `main` starts by allocating two pointers and storing them to variables `p` and `q`. The contents of both memory locations `p` and `q` are initialized to 0. Then, the value at location `p` is incremented nondeterministically often, and the value at location `q` is decremented nondeterministically often. The `assert` statements express that, at the end of the program the values in memory at `p` and `q` contain a non-negative or a non-positive value respectively.

As an intermediate goal to correctness, a solver must prove that the operations on memory cells `p` and `q` do not interfere. A typical CEGAR-based, or bounded model checking-based, solver will need to do this for every spurious counterexample.

```
var mem : [int] : int;
var valid : [int] : bool;

procedure main() {
  var p, q : int;

  call p := malloc();
  call q := malloc();

  mem[p] := 0;
  mem[q] := 0;

  while (*) {
    if (*) {
      mem[p] := mem[p] + 1;
    } else {
      mem[q] := mem[q] - 1;
    }
  }

  assert mem[p] >= 0;
  assert mem[q] <= 0;
}

procedure malloc() returns (ptr : int);
ensures !old(valid)[ptr];
ensures valid == old(valid)[ptr:=true];
```

```
var mem_1, mem_2 : [int] : int;
var valid : [int] : bool;

procedure main() {
  var p, q : int;

  call p := malloc();
  call q := malloc();

  mem_1[p] := 0;
  mem_2[q] := 0;

  while (*) {
    if (*) {
      mem_1[p] := mem_1[p] + 1;
    } else {
      mem_2[q] := mem_2[q] - 1;
    }
  }

  assert mem_1[p] >= 0;
  assert mem_2[q] <= 0;
}

procedure malloc() returns (p : int);
ensures !old(valid)[ptr];
ensures valid == old(valid)[p := true];
```

Fig. 1: Example of a program and its transformation. The program serves also as the basis of our scalable benchmark suite. — The value of the variable `mem` is a mathematical map. It is used to model the memory. The program transformation makes the independence of the two statements in the loop apparent. Intuitively, the two statements use the map `mem` differently. The transformation introduces *diffent maps for different uses.*

Our technique provides a preprocessing such that the solver can instead prove correctness of the transformed program on the right hand side of Figure 1. In the transformed example, the map `mem` has been replaced by two maps `mem_1` and `mem_2`. Memory accesses at `p` are modeled by accessing `mem_1`, memory accesses at `q` are modeled by accessing `mem_2`. That way the solver does not need to prove non-interference between the increment and decrement operations for each spurious counterexample, which typically results in a dramatic speedup.

3 Preliminaries

In this section, we fix our notation regarding program syntax and semantics.

Program Syntax We distinguish two types of variables, *map variables* and *base variables*. Map variables are named $\mathtt{a}, \mathtt{b}, \ldots$. We use $\mathtt{i}, \mathtt{j}, \ldots$ for base variables that are used as map indices in the current context and $\mathtt{x}, \mathtt{y}, \ldots$ for all-purpose base variables. We use constant (or literal) expressions named $\mathsf{lit}, \mathsf{lit}_1, \mathsf{lit}_2, \ldots$. We use a special variable $\mathtt{pc} \in \mathsf{Variables}$ called the *program counter*. We use typewriter font for program variables (e.g., `i`, `x`) and italics for mathematical variables (e.g., i,x).

Expressions in our programs can have one of three types.

Expressions of base type:	$e_{\mathsf{base}} ::= \mathsf{lit} \mid \texttt{x} \mid \texttt{a[i]}$
Expressions of map type:	$e_{\mathsf{map}} ::= \texttt{a} \mid \texttt{a[i:=x]} \mid \texttt{(const}\ \mathsf{lit}\texttt{)}$
Boolean expressions:	$e_{\mathsf{bool}} ::= \texttt{x==y} \mid \texttt{!}\, e_{\mathsf{bool}} \mid e_{\mathsf{bool}}\ \texttt{\&\&}\ e_{\mathsf{bool}} \mid e_{\mathsf{bool}}\ \texttt{||}\ e_{\mathsf{bool}}$

The set of all *commands* is generated generated by the following grammar. We refer to this set by $\mathsf{Commands}$.

$$c ::= \texttt{x:=}e_{\mathsf{base}} \mid \texttt{a:=}e_{\mathsf{map}} \mid \texttt{havoc x} \mid \texttt{havoc a} \mid \texttt{assume}\ e_{\mathsf{bool}}$$

The set of *program locations*, Loc, is a set of distinct identifiers $\{\ell, \ell', \ell_0, \ell_1, \ldots\}$. A *statement* is a triple of a source program location, a command, and a target program location, i.e., $\mathsf{Statements} = \mathsf{Loc} \times \mathsf{Commands} \times \mathsf{Loc}$. We use the letter σ for statements. Let $\sigma = (\ell, c, \ell')$ be a statement, then we refer to the *source location* of σ by $src(\sigma)$. In contexts where the locations are not important we omit them from the statement and write only the command. We call statements whose command is of the form `a:=a[i:=x]` *map write statements*, and we call statements whose command is of the form `x:=a[i]` *map read statements*. To highlight that a statement's command is a map write (read), we name the statement σ_{wr} (σ_{rd}).

A *program* P is given as a control flow graph whose edges are statements. Formally: $P = (\mathsf{Loc}, \Sigma, \ell_0)$, where Loc is a set of locations, $\Sigma \subseteq \mathsf{Statements}$ is a set of statements, and $\ell_0 \in \mathsf{Loc}$ is the initial location. For technical reasons we do not allow incoming control flow edges at the initial location. A program P induces a set of *program variables*, *Var*, which are all the variables that occur in any of the statements of P. We sometimes refer to only the basic variables $Var_{base} \subseteq Var$ or only the map variables $Var_{map} \subseteq Var$. We call the subset of Σ that contains all the map write (read) statements Σ_{wr} (Σ_{rd}). From now on we assume the program P is given as described here.

We do not allow equating maps in assume statements (`assume a==b`). In our experience this restriction does not matter in practice. Furthermore, we only allow equalities between (base) variables, not between expressions. This is not a proper restriction.

We will abbreviate `a:=a[i:=x]` as `a[i]:=x`. We may omit the case when the store is over a different map, like `a:=b[i:=x]`, from case distinctions, since it can be simulated by a map update followed by a map assigment; in this case `a:=b` followed by `a[i]:=x`. Also, we omit chains of stores applied to one map variable; again this omission does not change the expressiveness of the programming language.

Program Semantics For simplicity of presentation we consider only two sorts, namely the base sort *Sort* and the map sort $Sort \rightarrow Sort$.

A *state* in our program is a mapping from program variables to values from our set of sorts. The base variables, like `x` and `i` are assigned values of sort *Sort*.

The map variables, like a, are assigned values of sort $\mathit{Sort} \to \mathit{Sort}$. The Boolean sort $\{\texttt{true}, \texttt{false}\}$ occurs only during evaluation of Boolean expressions. The program counter variable pc is a special case, its value denotes the location $\ell \in \mathsf{Loc}$ that the execution is currently in.

We use the (semantic) map update operator $\cdot[\cdot \mapsto \cdot] \colon (\mathit{Sort} \to \mathit{Sort}) \times \mathit{Sort} \times \mathit{Sort} \to (\mathit{Sort} \to \mathit{Sort})$: Let a be a map, then $a[i \mapsto x]$ is the map that returns the value $a(j)$ for all arguments $j \neq i$ and the value x for the argument i.

For expressions e we give an evaluation function $\cdot[\![\cdot]\!] \colon \mathsf{States} \times \mathsf{Expressions} \to (\mathit{Sort} \cup (\mathit{Sort} \to \mathit{Sort}))$, which, given a valuation of the variables, assigns a value to e: Every literal has one value in *Sort* it is associated with; the literal evaluates to that value regardless of state. A variable is evaluated by looking up its value in the state. A map variable's value is a map, a map access at some index evaluates to the application of the evaluated map value to the evaluated index value. The semantics of the store operator is given as the above-mentioned map update operator. A constant map expression with some argument lit evaluates to a map whose value is lit at every position. The Boolean operators are evaluated as usual. Formally:

$$s[\![\mathsf{lit}]\!] \overset{def}{=} \mathsf{lit} \qquad s[\![\mathtt{v}]\!] \overset{def}{=} s(\mathtt{v})$$

$$s[\![\mathtt{a[i]}]\!] \overset{def}{=} s[\![\mathtt{a}]\!](s[\![\mathtt{i}]\!]) \qquad s[\![\mathtt{a[i:=x]}]\!] \overset{def}{=} s[\![\mathtt{a}]\!][s[\![\mathtt{i}]\!] \mapsto s[\![\mathtt{x}]\!]]$$

$$s[\![\mathtt{(const}\ \mathsf{lit}\mathtt{)}]\!] \overset{def}{=} \lambda x.\, \mathsf{lit} \qquad s[\![\mathtt{e==e'}]\!] \overset{def}{=} \begin{cases} \mathtt{true} & \text{if } s[\![\mathtt{e}]\!] = s[\![\mathtt{e'}]\!] \\ \mathtt{false} & \text{otherwise} \end{cases}$$

The *concrete post* operator $\mathsf{post} \colon 2^{\mathsf{States}} \times \mathsf{Statements} \to 2^{\mathsf{States}}$ is given as follows.

$$\mathsf{post}(S, (\ell,\ \mathtt{x:=}e_{\mathsf{base}}, \ell')) \overset{def}{=} \{s[\mathtt{pc} \mapsto \ell'][\mathtt{x} \mapsto s[\![e_{\mathsf{base}}]\!]] \mid s \in S, s(\mathtt{pc}) = \ell\}$$

$$\mathsf{post}(S, (\ell,\ \mathtt{a:=}e_{\mathsf{map}}, \ell')) \overset{def}{=} \{s[\mathtt{pc} \mapsto \ell'][\mathtt{a} \mapsto s[\![e_{\mathsf{map}}]\!]] \mid s \in S, s(\mathtt{pc}) = \ell\}$$

$$\mathsf{post}(S, (\ell,\ \mathtt{havoc\ x}, \ell')) \overset{def}{=} \{s[\mathtt{pc} \mapsto \ell'][\mathtt{x} \mapsto v] \mid s \in S, s(\mathtt{pc}) = \ell, v \in \mathit{Sort}\}$$

$$\mathsf{post}(S, (\ell,\ \mathtt{havoc\ a}, \ell')) \overset{def}{=} \{s[\mathtt{pc} \mapsto \ell'][\mathtt{a} \mapsto v] \mid s \in S, s(\mathtt{pc}) = \ell, v \in \mathit{Sort} \to \mathit{Sort}\}$$

$$\mathsf{post}(S, (\ell,\ \mathtt{assume\ e}, \ell')) \overset{def}{=} \{s[\mathtt{pc} \mapsto \ell'] \mid s \in S, s(\mathtt{pc}) = \ell, s[\![\mathtt{e}]\!] = \mathtt{true}\}$$

An *execution* e is a sequence of statements and states in alternation, i.e.,

$$e = s_0 .\, \sigma_0 . \ldots . \sigma_{n-1} .\, s_n .$$

Every execution starts in an initial state, i.e., a state s_0 where the program counter pc is assigned the initial location ℓ_0. Furthermore, the sequence must be consecutive, i.e., for all i from 0 to $n-1$, the state s_{i+1} must be contained in the set of post states of the state s_i under the statement σ_i, i.e.,

$$s_{i+1} \in \mathsf{post}(\{s_i\}, \sigma_i).$$

A special case are the empty executions, an empty execution s_0 consists of an initial state only. We can write every non-empty execution as $e.\sigma.s$ where e is an execution. We denote the set of all executions $\mathsf{Executions}$.

The *reachable states* are all states s such that there is an execution that ends in s.

$$Reach \overset{def}{=} \{s \mid \exists e \in \mathsf{Executions}.\, e = e's\}$$

4 Dependency Analysis

Our program transformation is based on an analysis of the dependencies between the statements in the program P. In this section, we describe a property that makes explicit which map update statements may be reponsible for the value of a map at some index at some program location. For this, we introduce the relation LstWr (read: "last writes") that contains for a potential read in the program all the map updates that are relevant for that read in some execution of the program.

Last Write Relation LstWr The relation $\mathsf{LstWr} \subseteq \Sigma_{\mathsf{wr}} \times \Sigma_{\mathsf{rd}}$ relates all map write statements σ_{wr} to all the map read statements σ_{rd} such that σ_{wr} is responsible for the value that is read in σ_{rd} in some execution.

Definition 1 (Last Writes Relation LstWr). *The Last Write relation* $\mathsf{LstWr} \subseteq \Sigma_{\mathsf{wr}} \times \Sigma_{\mathsf{rd}}$ *contains a pair* $(\sigma_{\mathsf{wr}}, \sigma_{\mathsf{rd}})$*, where the command in* σ_{wr} *is of the form* `a[i]:=x`*, and the command in* σ_{rd} *is of the form* `y:=b[j]`*, whenever there is an execution* e *and a value* v *such that* v *is written by* σ_{wr} *and is read by* σ_{rd}*, i.e., if* e *fulfills the following linear time property.*

$$\Diamond\,(pc = src(\sigma_{\mathsf{wr}}) \wedge x = v \wedge \Diamond\,(pc = src(\sigma_{\mathsf{rd}}) \wedge b\,[\,j\,] = v))$$

In this definition we assume that every value that is written to a map during an execution is unique; this can be accommodated by providing each value with a timestamp. Furthermore, in this definition a and b may or may refer to the same program variables, the same holds for, i and j and x and y.

Alternative Characterisation of the Last Writes Relation LstWr We provide an alternative characterisation of the Last Writes relation LstWr. This characterisation will lead to an instrumentation of the program that will allow us to compute an relation $\mathsf{LstWr}^{\#}$ that overapproximates the Last Writes relation.

We next define the function lw which, given a position i, given a map $\mathtt{a}$, and given an execution e, returns the write statement σ_{wr} that is responsible for the value that the map $\mathtt{a}$ has at position i in the last state of the execution e. For technical reasons we will use the symbol $\bot$ (to cater for the case where the map $\mathtt{a}$ has not been written at position i in execution e).

Formally, we define the function $\mathsf{lw}\colon Var_{\mathsf{map}} \times Sort \times \mathsf{Executions} \rightarrow \Sigma_{\mathsf{wr}} \cup \{\bot\}$ by induction over the length of the execution e. (As explained above, an execution of length 0 is of the form s_0 where s_0 is an initial state, and an execution of length $n+1$ is of the form $e.\,\sigma.\,s$ where σ is a statement and s is a state.)

$$
\begin{aligned}
\mathsf{lw}(\mathtt{a}, j, s_0) &\overset{def}{=} \bot \\
\mathsf{lw}(\mathtt{a}, j, e.\,\mathtt{havoc\ \ a}.\,s) &\overset{def}{=} \bot \\
\mathsf{lw}(\mathtt{a}, j, e.\,\mathtt{a:=(const\ \ }\mathsf{lit}\mathtt{)}.\,s) &\overset{def}{=} \bot \\
\mathsf{lw}(\mathtt{a}, j, e.\,\mathtt{a[i]:=x}.\,s) &\overset{def}{=} \begin{cases} \mathtt{a[i]:=x} & \text{if } s(\mathtt{i}) = j \\ \mathsf{lw}(\mathtt{a}, j, e) & \text{if } s(\mathtt{i}) \neq j \end{cases} \\
\mathsf{lw}(\mathtt{a}, j, e.\,\mathtt{a:=b}.\,s) &\overset{def}{=} \mathsf{lw}(\mathtt{b}, j, e) \\
\mathsf{lw}(\mathtt{a}, j, e.\,\sigma.\,s) &\overset{def}{=} \mathsf{lw}(\mathtt{a}, j, e) \text{ if } e.\,\sigma.\,s \text{ matches none of the above}
\end{aligned}
$$

Intuitively, the definition of $\mathsf{lw}(\mathtt{a}, j, e)$ traces the value of the map $\mathtt{a}$ at index j back within the execution e until it hits the map write statement that is responsible for the fact that $\mathtt{a}$ has that value at position j at the end of e. This write statement is returned by lw. If the execution consists only of an initial state s_0, or the last statement was a havoc statement with argument $\mathtt{a}$, or when $\mathtt{a}$ has been set to a constant map by the last statement, then no value in $\mathtt{a}$ depends on a map write statement, so lw returns the symbol $\bot$. If the last statement in the execution has been a write to map $\mathtt{a}$, then LstWr checks whether the write was at position j. If that is the case, the last write is returned, otherwise lw recurses on the prefix of the execution where the write statement and its successor state have been dropped. If the last statement in the execution assigned another map $\mathtt{b}$ to $\mathtt{a}$, the lw recurses on the execution prefix, and it looks for writes on $\mathtt{b}$ instead of writes on $\mathtt{a}$. Otherwise, the last statement in the execution had no influence on values in $\mathtt{a}$, so it is evaluated recursively on the prefix without the last statement and state.

As above, the Last Writes relation LstWr relates all the write statements σ_{wr} to all the read statements σ_{rd}, such that there is an execution where σ_{wr} is responsible for the value that σ_{rd} reads. From the function lw we build the explicit characterization of the relation $\mathsf{LstWr} \subseteq \Sigma_{\mathsf{wr}} \times \Sigma_{\mathsf{rd}}$ as follows.

$$
\begin{aligned}
\mathsf{LstWr} \overset{def}{=} \ &\{(\sigma_{\mathsf{wr}}, \sigma_{\mathsf{rd}}) \mid \sigma_{\mathsf{rd}} = (\ell, \mathtt{x:=a[i]}, \ell') \\
&\quad \wedge \exists\, e.\,s \in \mathsf{Executions}.\ s(\mathtt{pc}) = \ell \wedge s(\mathtt{i}) = i \wedge \mathsf{lw}(\mathtt{a}, i, e.\,s) = \sigma_{\mathsf{wr}} \\
&\quad \wedge \sigma_{\mathsf{wr}} \neq \bot\}
\end{aligned}
$$

5 Computing Dependencies

In this section, we present an instrumentation of the program P such that the Last Writes relation LstWr can be expressed in terms of the set of reachable states of the instrumented program P_{LstWr}.

5.1 Instrumentation

We introduce an auxiliary map variable `a-lw` for every map-variable `a` that occurs in the program P. The values of the maps that are assigned to `a-lw` are not values from our base sort *Sort*, but instead are symbols that refer to write statements that occur in P.

Intuitively, the transformation is designed in such a way that the fresh `lw`-maps capture the results of the lw-function for each program location. We construct the transformation in three steps. We begin by defining by a transformer $\tau^c_{\mathsf{LstWr}} \colon \mathsf{Commands} \to \mathsf{Commands}$ for some commands whose transformation result does not depend on their location in the program.

If the command c is a havoc to map variable `a`, or if c assigns a constant map to `a`, then `a-lw` is assigned a constant map that contains the symbol $\perp$ at all positions. This represents that no write statement has an influence on any value in the map `a` after the command c has been executed. If c assigns the value of a map variable to another map variable, then the analogous assignment is done on the respective `lw`-maps. This expresses that all map write statements that have an influence on `a` also have an influence on `b` after the command c has been executed. In all other cases, the transformation τ^c_{LstWr} leaves the command c unchanged.

$$\begin{aligned}
\tau^c_{\mathsf{LstWr}}(\texttt{havoc a}) &\overset{\mathit{def}}{=} \texttt{havoc a; a-lw:=(const }\perp\texttt{)} \\
\tau^c_{\mathsf{LstWr}}(\texttt{a:=(}\mathsf{const\ \ lit}\texttt{)}) &\overset{\mathit{def}}{=} \texttt{a:=(}\mathsf{const\ \ lit}\texttt{); a-lw:=(const }\perp\texttt{)} \\
\tau^c_{\mathsf{LstWr}}(\texttt{b:=a}) &\overset{\mathit{def}}{=} \texttt{b:=a; b-lw:=a-lw} \\
\tau^c_{\mathsf{LstWr}}(c) &\overset{\mathit{def}}{=} c \text{ where none of the other cases apply}
\end{aligned}$$

From τ^c_{LstWr} we construct the transformer $\tau^\sigma_{\mathsf{LstWr}} \colon \mathsf{Statements} \to \mathsf{Statements}$, which transforms the map write statements. Whenever a map variable `a` is written to at index `i`, then `a-lw` is written at the same index, but with a special value that identifies the updating statement. Statements that are not map write statements are left unchanged by $\tau^\sigma_{\mathsf{LstWr}}$.

$$\begin{aligned}
\tau^\sigma_{\mathsf{LstWr}}(\sigma_{\mathsf{wr}}) &\overset{\mathit{def}}{=} (\ell,\, \texttt{a[i]:=x; a-lw[i]:=}\sigma_{\mathsf{wr}}\,, \ell') \\
&\qquad \text{where } \sigma_{\mathsf{wr}} = (\ell,\, \texttt{a[i]:=x}\,, \ell') \\
\tau^\sigma_{\mathsf{LstWr}}((\ell, c, \ell')) &\overset{\mathit{def}}{=} (\ell, \tau_{\mathsf{LstWr}}(c), \ell') \text{ where } (\ell, c, \ell') \notin \Sigma_{\mathsf{wr}}
\end{aligned}$$

The final statement transformer $\tau_{\mathsf{LstWr}} \colon \mathsf{Statements} \to \mathsf{Statements}$ updates statements that originate from the initial location ℓ_0. Because at the initial location no map writes have been executed, we set every `lw`-variable to a constant map containing the symbol $\perp$. (Note that we assume that the initial location has no incoming statements.)

$$\begin{aligned}\tau_{\mathsf{LstWr}}(\sigma) \overset{def}{=}\ & c\texttt{;}\\ & \texttt{a-lw:=(const }\bot\texttt{);}\\ & \dots\\ & \texttt{z-lw:=(const }\bot\texttt{)}\\ & \text{ where } \tau^{\sigma}_{\mathsf{LstWr}}(\sigma) = (\ell_0, c, \ell) \text{ and } \mathit{Var}_{\mathsf{map}} = \{\texttt{a}, \dots, \texttt{z}\}\end{aligned}$$

We are now ready to define the instrumented program P_{LstWr}. We define the instrumented program P_{LstWr} through applying the transformation function τ_{LstWr} to each statement in Σ. Formally:

$$P_{\mathsf{LstWr}} \overset{def}{=} \{\mathsf{Loc}, \{\tau_{\mathsf{LstWr}}(\sigma) \mid \sigma \in \Sigma\}, \ell_0\}$$

We can now express the Last Write relation LstWr through the set of reachable states of the instrumented program P_{LstWr}.

Proposition 1. *The Last Writes relation* LstWr *as defined in Section 4 is identical to the relation that relates a map write statement* σ_{wr} *in* Σ_{wr} *to a map read statement* σ_{rd} *in* Σ_{rd} *of the form* $(\ell, \texttt{x:=a[i]}, \ell')$ *if there is a state* s *in the set of reachable states of the instrumented program* P_{LstWr} *such that the program counter* `pc` *points to the source location of* σ_{rd}*,* ℓ*, and the value that* s *assigns to the map read expression* `a-lw[i]` *is the write statement* σ_{wr}*. Formally:*

$$\begin{aligned}\mathsf{LstWr} = \{(\sigma_{\mathsf{wr}}, \sigma_{\mathsf{rd}}) \mid\ & \sigma_{\mathsf{rd}} = (\ell, \texttt{x:=a[i]}, \ell')\\ & \wedge \exists s \in \mathit{Reach}(P_{\mathsf{LstWr}}).\, s[\![\texttt{pc}]\!] = \ell \wedge s[\![\texttt{a-lw[i]}]\!] = \sigma_{\mathsf{wr}}\}\end{aligned}$$

We state the following lemma for later reference (proof of Theorem 1 in Section 6).

Lemma 1. *P and P_{LstWr} are bisimulation-equivalent.*

The proof of this lemma is obvious form the fact that the additional commands introduced by the transformation is ghost code.

5.2 Computing an Overapproximation of the Last Writes Relation LstWr

We have seen that the relation LstWr can be expressed through the set of reachable states of the instrumented program P_{LstWr}. The set of reachable states is not computable in general. Thus, we apply a static analysis that computes an overapproximation of the set of reachable states.

The static analysis must be able to handle programs that manipulate maps. An example is a static analysis based on the Map Equality Domain [4]. This domain is useful to infer equalities and disequalities between expressions which can involve maps.

We have implemented an extension of the Map Equality Domain. The extensions supports constraints of the form $\texttt{x} \in \{\mathsf{lit}_1, \mathsf{lit}_2\}$ which allows us to succinctly express constraints like $\texttt{a-lw[i]} \in \{\sigma_1, \sigma_2\}$. Here, σ_1 and σ_2 are literals (referring to the corresponding statements). All literals are pairwise different. Thus, these constraints allow us to infer constraints like $\texttt{a-lw[i]} \neq \sigma_3$. Such constraints are crucial to infer independence of statements.

From now on, we use $\mathsf{LstWr}^{\#}$ to refer to the overapproximation of the relation LstWr computed by applying the above-described static analysis to the instrumented program P_{LstWr}. The static analysis always computes an overapproximation of the set of reachable states of P_{LstWr}. Thus, the relation $\mathsf{LstWr}^{\#}$ is an overapproximation of the Last Writes relation LstWr. We state the following remark for later reference (in Lemma 2).

Remark 1. The relation $\mathsf{LstWr}^{\#}$ is an overapproximation of the Last Write relation LstWr, i.e.,

$$\mathsf{LstWr}^{\#} \supseteq \mathsf{LstWr}.$$

6 Program Transformation

In this section we introduce the program transformation that transforms the program P, given the relation $\mathsf{LstWr}^{\#}$, which approximates the Last Write relation LstWr of program P.

6.1 Computing a Partition of the Map Write Statements

First, we define the relation $R \subseteq \Sigma_{\mathsf{wr}} \times \Sigma_{\mathsf{wr}}$ that relates all write statement that map influence the same read statement. Two write statements σ_{wr} and σ'_{wr} are related by R if there exists a read statement σ_{rd} such that the relation $\mathsf{LstWr}^{\#}$ relates both σ_{wr} to σ_{rd} and σ'_{wr} to σ_{rd}. Formally:

$$R \overset{def}{=} \{(\sigma_{\mathsf{wr}}, \sigma'_{\mathsf{wr}}) \mid \exists\, \sigma_{\mathsf{rd}} \in \Sigma_{\mathsf{rd}}.\, \mathsf{LstWr}^{\#}(\sigma_{\mathsf{wr}}, \sigma_{\mathsf{rd}}) \wedge \mathsf{LstWr}^{\#}(\sigma_{\mathsf{wr}}, \sigma_{\mathsf{rd}})\}$$

Based on the relation R, we define the relation $r \subseteq \Sigma_{\mathsf{wr}} \times \Sigma_{\mathsf{wr}}$ as the smallest equivalence relation that contains the relation R. This equivalence relation r induces a partition over the set Σ_{wr}, i.e., a set $\mathcal{W} \subseteq 2^{\Sigma_{\mathsf{wr}}}$ of subsets of the set Σ_{wr} such that the disjoint union of the subsets is identical to the original set Σ_{wr}. Thus, the set $\mathcal{W}$ consists of disjoint subsets $\{W_1, \ldots, W_n\}$ of the set of all write statements Σ_{wr}. The partition $\mathcal{W}$ has the property that for every two blocks W_1 and W_2 in $\mathcal{W}$, we know that if we take one write statement σ_{wr} from W_1 and another write statement σ'_{wr} from W_2, then σ_{wr} and σ'_{wr} are independent in the sense that they never have an influence on the same read statement.

For technical reasons, we add a the singleton consisting only of the symbol $\perp$ to $\mathcal{W}$. Its use will become clear in the next subsection.

6.2 Program Transformation

We introduce a map variable $\texttt{a\_}W$ for each $W \in \mathcal{W}$. If for example the write statements `a[i]:=x` and `a[j]:=y` appear in different blocks W_1 and W_2, then we will replace the map variable `a` with two different variables $\texttt{a\_}W_1$ and $\texttt{a\_}W_2$ in these statements accordingly. (There is a subtle point here regarding the fact that W is a mathematical object while a variable name consists of characters which we neglect here.)

We use the notation $\mathsf{LstWr}^{\#-1}[\sigma_{\mathsf{rd}}]$ to denote the preimage of $\mathsf{LstWr}^{\#}$ with respect to some read statement $\sigma_{\mathsf{rd}} \in \Sigma_{\mathsf{rd}}$, i.e.,

$$\mathsf{LstWr}^{\#-1}[\sigma_{\mathsf{rd}}] \overset{def}{=} \{\sigma_{\mathsf{wr}} \mid (\sigma_{\mathsf{wr}}, \sigma_{\mathsf{rd}}) \in \mathsf{LstWr}^{\#}\}.$$

The transformation updates the statements of program P using the transformation $\tau\colon \mathsf{Statements} \to \mathsf{Statements}$ as described in the following. The transformation result $\tau(\sigma)$ depends on the statement type of σ. If σ writes to map variable `a`, it is transformed to a statement that does the same update to map variable $\texttt{a\_}W$, i.e., to the map variable corresponding to the block in the partition $W \in \mathcal{W}$ that contains σ. If σ reads from a map variable `a`, there are two cases. Either $\mathsf{LstWr}^{\#}$ at the read location yields the empty set. This means that it is guaranteed that the read position has never been written to in any execution that reaches σ. In this case, σ is transformed to a read from the map variable $\texttt{a\_}\{\bot\}$ instead of `a`. Otherwise, by construction of the partition $\mathcal{W}$, LstWr must yield a set that falls completely into a block W in the partition $\mathcal{W}$. In that case, σ is transformed to a read from the map variable $\texttt{a\_}W$ instead of `a`. If σ assigns a map variable `a` to a map variable `b`, it is transformed to a series of assignments that assign for each block in the partition $W \in \mathcal{W}$ the variable $\texttt{a\_}W$ to the variable $\texttt{b\_}W$. A havoc to a map variable `a` is translated to havoc on all variables $\texttt{a\_}W$ for every block W in the partition $\mathcal{W}$, followed by an assume statement that ensures that all maps $\texttt{a\_}W$ have been set to the same value. In all other cases, the transformation leaves σ unchanged. Formally:

$$
\begin{aligned}
\tau((\ell, \texttt{a[i]:=x}, \ell')) &\overset{def}{=} (\ell, \texttt{a\_}W\texttt{[i]:=x}, \ell') \text{ where } (\ell, \texttt{a[i]:=x}, \ell') \in W \\
\tau((\ell,\ \texttt{x:=a[i]}, \ell')) &\overset{def}{=} (\ell,\ \texttt{x:=a\_}\{\bot\}\texttt{[i]}, \ell') \\
&\qquad \text{if } \mathsf{LstWr}^{\#-1}[(\ell,\ \texttt{x:=a[i]}, \ell')] = \emptyset \\
\tau((\ell,\ \texttt{x:=a[i]}, \ell')) &\overset{def}{=} (\ell,\ \texttt{x:=a\_}W\texttt{[i]}, \ell') \\
&\qquad \text{if } \mathsf{LstWr}^{\#-1}[(\ell,\ \texttt{x:=a[i]}, \ell')] \neq \emptyset \\
&\qquad \text{and } \mathsf{LstWr}^{\#-1}[(\ell,\ \texttt{x:=a[i]}, \ell')] \subseteq W \\
\tau((\ell,\ \texttt{b:=a}, \ell')) &\overset{def}{=} (\ell,\ \texttt{b\_}W_1\texttt{:=a\_}W_1\texttt{;}\ \ \texttt{...;}\ \ \texttt{b\_}W_n\texttt{:=a\_}W_n, \ell') \\
&\qquad \text{where } \mathcal{W} = \{W_1, \dots, W_n\} \\
\tau(\sigma) &\overset{def}{=} \sigma \text{ if } \sigma \text{ matches none of the above cases}
\end{aligned}
$$

We construct the transformed program P' by replacing all statements σ in P by their transformed version $\tau(\sigma)$. Formally:

$$P' \overset{def}{=} (\mathsf{Loc}, \{\tau(\sigma) \mid \sigma \in \Sigma\}, \ell_0)$$

6.3 Correctness of the Transformation

In this subsection, we show that the transformation is correct, i.e., that the program P and the transformed program P' are bisimulation-equivalent. Given Lemmma 1, it is sufficient to prove the following Lemma.

As an aside: it does not seem obvious to us how to give a bisimulation between the programs P and P' directly.

Lemma 2. *The programs P_{LstWr} and P' are bisimulation-equivalent.*

Proof. We define a bisimulation relation $\sim$ between P_{LstWr} and P' as follows.

The states $s \in \mathsf{States}_P$ and $t \in \mathsf{States}_{P'}$ are bisimilar, i.e., $s \sim t$, iff

$$\forall x \in Var_{base}. \qquad s[\![x]\!] = t[\![x]\!] \qquad (1)$$

and

$$\forall \mathtt{a} \in Var_{map}. \forall \mathtt{i} \in Var_{base}.$$
$$(s[\![\mathtt{a-lw[i]}]\!] = \bot \implies \forall W \in \mathcal{W}. s[\![\mathtt{a[i]}]\!] = t[\![\mathtt{a\_}W\mathtt{[i]}]\!]) \quad (2a)$$
$$\wedge (\exists W \in \mathcal{W}. s[\![\mathtt{a-lw[i]}]\!] \in W \implies s[\![\mathtt{a[i]}]\!] = t[\![\mathtt{a\_}W\mathtt{[i]}]\!]) \quad (2b)$$

We show, that $\sim$ is a bisimulation. Pick s, t such that $s \sim t$ (We call this the induction hypothesis, I.H.). Pick σ in Σ_P (which corresponds to picking $\tau_{\mathsf{LstWr}}(\sigma)$ and $\tau(\sigma)$ as well).

We make a case distinction on which statement type σ falls into.

Case σ is an assignment: Let $\{s'\} \in \mathsf{post}(\{s\}, \tau_{\mathsf{LstWr}}(\sigma))$ and let $\{t'\} = \mathsf{post}(\{t\}, \tau(\sigma))$.

First, we consider the conditions (1), (2a), and (2b) with respect to variables x, a, and i that are not updated by σ when σ is deterministic. For all three conditions, the reasoning is simple: By I.H. the condition holds with respect to s and t. Neither $\tau_{\mathsf{LstWr}}(\sigma)$ nor $\tau(\sigma)$ modify x, a or i as they occur in the conditions, and $\tau_{\mathsf{LstWr}}(\sigma)$ does not modify a-lw. Thus the conditions directly carry over from s and t to s' and t'.

In order to prove the conditions for s' and t' with respect to to variables that are updated by σ, we make a further case distinction on which type of assignment σ is (and analogously by $\tau_{\mathsf{LstWr}}(\sigma)$ and $\tau(\sigma)$).

- Case $\sigma = \mathtt{a[i]:=x}$: a is updated only at position $s'[\![\mathtt{i}]\!]$; for the other positions, the same reasoning as above is applicable. We know $s'[\![\mathtt{a-lw[i]}]\!] = \sigma$ and $\sigma \neq \bot$. Thus the antecedent of condition (2a) cannot be fulfilled in s'. Let $W \in \mathcal{W}$ be the block that contains σ. Remember $\tau(\sigma) = \mathtt{a\_}W\mathtt{[i]:=x}$ and $\tau_{\mathsf{LstWr}}(\sigma) = \mathtt{a[i]:=x;}\ \ \mathtt{...}$. Thus $t'[\![\mathtt{a\_}W\mathtt{[i]}]\!] = t[\![\mathtt{x}]\!] = s[\![\mathtt{x}]\!] = s'[\![\mathtt{a[i]}]\!]$, which means condition (2b) is fulfilled.

- Case $\sigma = \texttt{x:=a[i]}$: Then $\tau(\sigma) = \texttt{x:=a\_}W\texttt{[i]}$ for some $W \in \mathcal{W}$. In order to show $s'[\![\mathsf{x}]\!] = t'[\![\mathsf{x}]\!]$, we need to show $s[\![\texttt{a[i]}]\!] = t[\![\texttt{a\_}W\texttt{[i]}]\!]$.

  First, if $W = \{\bot\}$, by construction of τ, we have $\mathsf{LstWr}^{\#}(\texttt{a}, \texttt{i}, s(\texttt{pc})) = \{\bot\}$. Thus, by Proposition 1 and Remark 1, we have $s[\![\texttt{a-lw[i]}]\!] = \bot$. Thus, by condition (2a) in I.H. we get $s[\![\texttt{a[i]}]\!] = t[\![\texttt{a\_}W\texttt{[i]}]\!]$.

  Second, if $W \neq \{\bot\}$, by construction of τ, we have $\mathsf{LstWr}^{\#}(\texttt{a}, \texttt{i}, s(\texttt{pc})) \setminus \{\bot\} \subseteq W$. Thus, by Proposition 1 and Remark 1, we have $s[\![\texttt{a-lw[i]}]\!] \in W$. Thus, by condition (2b) in I.H. we get $s[\![\texttt{a[i]}]\!] = t[\![\texttt{a\_}W\texttt{[i]}]\!]$.
- Case $\sigma = \texttt{b:=a}$: We must show conditions (2a) and (2b) holds for s' and t' for variable b and b-lw. We already showed this for a and a-lw above (because a is updated by $\tau_{\mathsf{LstWr}}(\sigma)/\tau(\sigma)$). Our proof goal follows directly from the fact that $s'(\texttt{a}) = s'(\texttt{b})$ and $s'(\texttt{a-lw}) = s'(\texttt{b-lw})$ and for all $W \in \mathcal{W}$, $t'(\texttt{a\_}W) = t'(\texttt{b\_}W)$ hold, which is ensured by the assignments in the statements $\tau_{\mathsf{LstWr}}(\sigma)$ and $\tau(\sigma)$.
- Case $\sigma = \texttt{x:=e}$: where e is not a map read. Then, we know $\tau_{\mathsf{LstWr}}(\sigma) = \tau(\sigma) = \sigma$. By I.H., condition (1), $s'[\![\mathsf{e}]\!] = t'[\![\mathsf{e}]\!]$ holds, because e is a base variable or a literal. Our goal $s'[\![\mathsf{x}]\!] = t'[\![\mathsf{x}]\!]$ follows directly.

Case σ is a havoc statement:

- Case $\sigma = \texttt{havoc a}$: We show show the simulation directions separately. First let $s' \in \mathsf{post}(s', \tau_{\mathsf{LstWr}}(\sigma))$. We need to show existence of an appropriate $t' \in \mathsf{post}(t', \tau(\sigma))$. Given $s'[\![\mathsf{a}]\!]$, pick $t'[\![\texttt{a\_W}]\!]$ for all Ws identical to that. (Clearly, this state t' is not blocked by the assume statement in $\tau(\sigma)$.)
  For the other simulation direction let $t' \in \mathsf{post}(t', \tau(\sigma))$. We need to show existence of an appropriate $s' \in \mathsf{post}(s', \tau_{\mathsf{LstWr}}(\sigma))$. We know that for all W, W', $t'[\![a_W]\!] = t'[\![a_{W'}]\!]$ holds (ensured by the assume statement in $\tau(\sigma)$). Pick $s'[\![\mathsf{a}]\!]$ such that it equals all the $t'[\![\texttt{a\_W}]\!]$.
- Case $\sigma = \texttt{havoc x}$: We can clearly choose the appropriate s' or t' such that condition (1) is met.

Case $\sigma = \texttt{assume}\ e_{\mathsf{bool}}$: Remember we did not allow the use of map variables in assume statements, so $\tau_{\mathsf{LstWr}}(\sigma) = \tau(\sigma) = \sigma$. Because of I.H., condition (1), s and t agree on all base variables. Thus $s[\![e_{\mathsf{bool}}]\!] = t[\![e_{\mathsf{bool}}]\!]$. Thus whenever an s' is not blocked by $\tau_{\mathsf{LstWr}}(\sigma)$, it is not blocked by $\tau(\sigma)$ and vice versa. □

Theorem 1 (Bisimulation). *P and P' are bisimulation-equivalent.*

Proof. This follows by transitivity of bisimulation-equivalence from Lemmas 1 and 2. □

7 Implementation in Ultimate

The purpose of this paper is to provide formal foundations of a program transformation that makes independence of groups of map accessing statements explicit and to prove it correct. However, we find it important to explant that the approach extends to a full fledged intermediate language.

We implemented our program transformation in the Ultimate program analysis framework[1]. The intermediate representation we support is the most expressive one used by Ultimate, namely the so-called *interprocedural control flow graph* (short: ICFG). ICFGs are control flow graphs whose edges are labeled with ith transition formulas. Transition formulas are arbitrary logical formulas over some background theory that contain an in- and an out-version for each program variable. Furthermore, ICFGs allow dedicated edges for procedure calls and returns. In the following we highlight the most important features that the programming language used so far does not have and explain what is necessary to support them.

Multidimensional Maps In order to support maps of higher dimensions, we need to slightly adapt the relation LstWr and the corresponding analysis. On a technical level this is done by having not one but several `lw`-maps for each map variable in the original program. For an n-dimensional map variable `a` we would introduce n `lw`-maps `a-lw-1` to `a-lw-n` where `a-lw-1` is one-dimensional `a-lw-2` is two-dimensional and so forth.

Transition Formulas In transition formulas, the distinction between assume statements and assignments is not immediately apparent. For example, given a program variable `a`, the transition formula `a'` $= 1$ would correspond to the assignment `a:=1`, while the transition formula `a` $= 1 \wedge a' = a$ would correspond to the assume statement `assume a==1`. In order to infer, how our instrumentation needs to be done, we need to compute, which which variables are unconstrained in a given formula. Those have to be treated like variables subject to a havoc statements are treated.

Procedures In order to support procedures, two features are relevant: Map-valued parameters must be passed between procedures, and it must be possible to compute procedure summaries that describe the effect of a procedure on global map variables (in fact having one of these features would be enough in terms of expressiveness, but Ultimate supports both). Both of these features are enabled by our support for (by-value) assignments between maps.

8 Experiments on a Scalable Benchmark Suite

The thorough experimentation needed to establish whether the approach can be made applicable to classes of practical benchmarks (or, to what classes) is not in the scope of this paper. In this section, we will only investigate whether the approach is applicable in principle. That is, we will use a benchmark suite which is specifically tailored to condensate the case split explosion problem. This helps

[1] `https://github.com/ultimate-pa/ultimate`

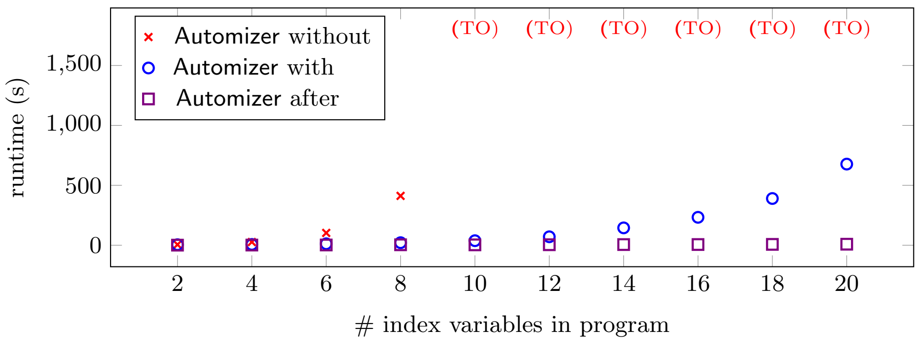

Fig. 2: The Ultimate Automizer toolchain *without* and *with* the program transformation as a preprocessing step, and the Ultimate Automizer toolchain in isolation applied *after* the program transformation, on a benchmark suite whose programs are scaled-up versions of the example program in Section 2. The timeout (TO) is set to 1800 seconds.

us to factor out all aspects in automatic program verification that are orthogonal to our problem.

We obtain the bechmark suite by starting with the example program from Section 2. The example program manipulates the map variable *mem* on the two *index* variables p and q. We obtain a new program by adding another two variables and adding the corresponding statements which manipulate the map variable *mem* on two new variables in the same way as the existing statements do for p resp. q. We can iterate the process and thus obtain a scalable benchmark suite whose programs have 2, 4, 6, ... index variables.

Setup We ran our experiments on a dedicated benchmarking system, each benchmark task was limited to 2 CPU cores at 2.4GHz and 20 Gigabytes of RAM. We ran two toolchains and took three measurements. One toolchain, called "Automizer without", is the standard verification toolchain of the program verifier Ultimate Automizer. The toolchain computes an ICFG from the input program and then run's Automizer's verification algorithm on the ICFG. The second toolchain, called "Automizer with", applies our transformation after computing the interprocedural control flow graph and before running Automizer's verification algorithm. A third kind of measurements, denoted "Automizer after", are the timings of only the verification algorithm in the toolchain "Automizer with", i.e., how long the verification of the transformed program takes.

Results In Figure 2 we display the results of our experimental evaluation. The x-axis of the plot represents the different example programs, identified by the number of map index variables. The y-axis represents the time taken by each toolchain. We ran three toolchains: The Ultimate Automizer program verifier, Ultimate Automizer where before the verification run, the transformation is applied, and a toolchain where Automizer was run on the already transformed programs.

We observe that the timings of Automizer on the transformed programs are nearly constant in the number of used map index variables – the timings range from 0.9 seconds to 8.8 seconds. This means that the only real difficulty in our programs lies in deriving the non-interferences between the map accesses. Furthermore, we can see that the Automizer fails to scale well when it needs to derive the non-interferences itself: It fails to prove all examples with 10 or more map index variables. The toolchain that includes our transformation shows a significantly improved scaling behaviour even though the transformation (in particular the static analysis it is based on) is not cheap.

9 Related Work

There are several works resembling ours in that they propose computing non-interference properties between memory regions to simplify the verification conditions that are handed to an SMT solver. Rakamaric and Hu [14], as well as Wang et al. [15] propose a memory model that uses maps which are separated according to the results of an up-front alias analysis. Gurfinkel and Navas [9] propose a related but different memory model. In their setting, the heap state is passed between procedures through local map variables. They propose a memory model with a partitioning that is context-sensitive to improve precision. In contrast to our work, these papers all rely on C semantics for their input program, so they do not apply to arbitrary map manipulating programs.

Our relation LstWr and the corresponding property is reminiscent of a large field of work that is concerned with inferring guarantees about data dependencies between program parts in the presence of arrays. We can only mention a few papers here, e.g., [5,13,12]. These papers propose various approaches of finding data dependencies in programs with arrays in different precisions, for different fragments and for different applications. None of them is aimed at symbolic program verification as our work is. To our knowledge, our property is the only one that accounts for maps, the crucial difference being the presence of by-value assignments.

10 Discussion

We discuss some of the choices we made in this paper.

10.1 Alias Analysis vs. Intermediate Verification Languages

In this subsection, we discuss why classical alias analyses cannot be used as a basis for our program transformation.

A classical alias analysis reasons about the pointer variables of a program. In a nutshell, the analysis collects all the assignments in the program that assign a

```
// (memory model infrastructure)
procedure main() {
  var p, q : int;
  p = 0;
  q = 0;
  // (code not using mem[p] or mem[q])
  p = malloc();
  q = malloc();
  // (code using mem[p] and mem[q])
}
```

Fig. 3: Program that illustrates why it is not sufficient to only consider pointer (map index) variables in our setting. Without any additional assumptions we must conclude that `p` and `q` may alias and thus that there is a dependency between statements that use `p` and `q` to access the map `mem`. However, if we consider at which program locations `p` and `q` are actually used to access the map `mem`, we can conclude that those accesses must be mutually independent. (An ensures statement guarantees that the procedure `malloc` never returns the same value twice.)

pointer source value to a pointer variable. Possible source values are typically: (1) calls to memory-allocating procedures, like `malloc`, (2) expressions that point to memory that is known to be implicitly allocated, like the addressof-expression `&x`, (3) other pointers. While the classes of source values may vary, it is always assumed that no two pointers alias "by accident". I.e., when a pointer is uninitialized, it is assumed to be distinct from every other pointer, even though nothing is known about its value at the time. The same holds for pointers that have been freed. Similarly, every pointer that has the value `null` is assumed to not alias with any other pointer, even if that other pointer also has the value `null`. To summarize, only *valid* pointer values are taken into account for alias analysis. This is sound in the context of the programming language because accessing an invalid pointer would lead to undefined behaviour according to the language standard. Thus, the analysis reasons about pointers with the hidden assumption that no undefined behaviour occurs in the program because in the case of undefined behaviour all guarantees about what the program does are lost anyways.

These assumptions enable extremely efficient pointer analyses because in this setting the only way that two pointers can alias is if there is a chain of assignments between pointer variables that (transitively) assigns the value of one pointer variable to the other. Therefore, a flow-insensitive analysis that collects all assignments of pointer variables without regard to control flow can already achieve good precision while being highly scalable.

The analogue to pointers in an intermediate verification language are map indices, i.e., values that are used to read values from a map variable. It is common to use mathematical integers as the sort of map indices, like in our example. In our setting, assumptions that are not explicitly modelled in the program are not allowed. Therefore, we have two options: (1) We model all assumptions in our verification language. E.g. we would have to check that all pointer accesses are indeed valid. This is impractical as checking this is a hard verification task on its own right. (2) We develop an alternative to alias analysis that does not rely on these assumptions – which is what we did in this paper.

As the example program in Figure 3 illustrates, it is not enough if our static analysis only considers the values that map indices may assume. Instead, we must track when and how (for read or write accesses) the indices are actually used. This is done by the Last Writes relation LstWr.

10.2 Assume Statements over Map Variables

From a theoretical of view, it might be interesting why we omit assume statements that equate map variables from our programming language. We now explain the complications this would entail.

Consider the following program snippet.

```
x := b[i];
a[j]:=y;
assume a==b;
```

The snippet contains no loops or procedure calls but still the map write in the second line influences the map read that comes *earlier* in the code because the assume statement establishes a relationship between the maps `a` and `b`. Thus, because `i` and `j` may alias, we have $(\texttt{a[j]:=y}, \texttt{x:=b[i]}) \in \mathsf{LstWr}$ (note that the assume statement enforces the timestamps to match as well as the values, between `a` and `b`). This would mean that a practical computation of LstWr would have to incorporate both forward- and backward analysis, whereas without such assume statements it is sufficient to propagate information in just one direction.

11 Conclusion

We have investigated the theoretical foundations for a novel research question which may be relevant for the practical potential of intermediate verification languages. The question concerns a preprocessing step for intermediate verification languages which takes the similar role that alias analysis plays in the verification for programming languages. We have presented a preliminary solution in the form of a program transformation. We have integrated the program transformation into a toolchain. A preliminary experimentation shows that the program transformation can be effective, at least in principle. On a benchmark suite which is specifically tailored to condensate the case split explosion problem, the toolchain with the program transformation scales very well in the size of the program (whereas the toolchain without the program transformation quickly falls into the case explosion problem and runs out of time or space).

The thorough experimentation needed to establish whether the approach can be made applicable to classes of practical benchmarks (or, to what classes) is not in the scope of this paper. We see our investigation as a preliminary for a wealth of future investigations to explore the practical potential of intermediate verification languages.

References

1. CHC-comp. `https://chc-comp.github.io/`.
2. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87. ACM, 2005.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309. ACM, 2007.
4. D. Dietsch, M. Heizmann, J. Hoenicke, A. Nutz, and A. Podelski. The map equality domain. In *VSTTE (to appear)*, Lecture Notes in Computer Science. Springer, 2018.
5. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
6. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
7. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
8. M. Greitschus, D. Dietsch, and A. Podelski. Loop invariants from counterexamples. In *SAS*, volume 10422 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2017.
9. A. Gurfinkel and J. A. Navas. A context-sensitive memory model for verification of C/C++ programs. In *SAS*, volume 10422 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 2017.
10. R. Leino. This is Boogie 2. Microsoft Research, June 2008.
11. J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
12. Y. Paek, J. Hoeflinger, and D. A. Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.
13. W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *LCPC*, volume 768 of *Lecture Notes in Computer Science*, pages 546–566. Springer, 1993.
14. Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *VM-CAI*, volume 5403 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2009.
15. W. Wang, C. Barrett, and T. Wies. Partitioned memory models for program analysis. In *VMCAI*, volume 10145 of *Lecture Notes in Computer Science*, pages 539–558. Springer, 2017.