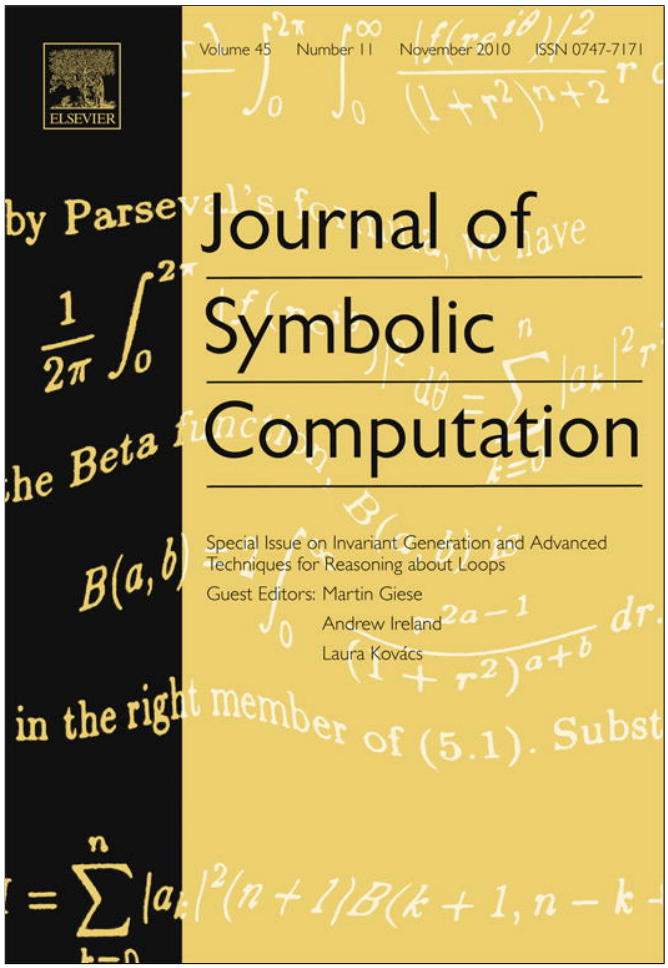
Modular inference of subprogram contracts for safety checking

Yannick Moy, Claude Marché



Journal of Symbolic Computation 45 (2010) 1184–1211

Contents lists available at ScienceDirect

Journal of Symbolic Computation

journal homepage: www.elsevier.com/locate/jsc

Modular inference of subprogram contracts for safety checking☆

Yannick Moy [a,1], Claude Marché [b,c]

[a] AdaCore - 46 rue d'Amsterdam, F-75009 Paris, France
[b] INRIA Saclay - Île-de-France, 4 rue Jacques Monod, F-91893 Orsay, France
[c] LRI, CNRS & Univ. Paris-Sud 11, F-91405 Orsay, France

ARTICLE INFO

Article history:
Received 15 March 2010
Accepted 11 April 2010
Available online 25 June 2010

Keywords:
Specification languages
Functional behavior
Contracts
Inference
Abstract interpretation
Weakest precondition
Quantifier elimination

ABSTRACT

Contracts expressed by logic formulas allow one to formally specify expected behavior of programs. But writing such specifications manually takes a significant amount of work, in particular for *uninteresting* contracts which only aim at avoiding run-time errors during the execution. Thus, for programs of large size, it is desirable to at least partially infer such contracts. We propose a method to infer contracts expressed as boolean combinations of linear equalities and inequalities by combining different kinds of static analyses: abstract interpretation, weakest precondition computation and quantifier elimination. An important originality of our approach is to proceed modularly, considering subprograms independently.

The practical applicability of our approach is demonstrated on experiments performed on a library and two benchmarks of vulnerabilities of C code.

1. Introduction

Behavioral interface specification languages allow one to specify complex properties of the functional behavior of programs. For a survey, see Hatcliff et al. (2009). Examples of such languages are JML (Burdy et al., 2004) for Java, Spec# (Barnett et al., 2004) for C#, and ACSL (Baudin et al., 2008)

☆ This research was partly supported by the French national projects: CAT (*C Analysis Toolbox*, ANR-05-RNTL); CIFRE PhD contract 2005/973 with France Telecom; and U3CAT (*Unification of Critical C Code Analysis Techniques*, ANR-09-ARPEGE).

E-mail addresses: yannick.moy@gmail.com (Y. Moy), Claude.Marche@inria.fr (C. Marché).
[1] Tel.: +33 1 72 92 59 69; fax: +33 1 74 85 42 29.

0747-7171/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2010.06.004

and VCC (Dahlweid et al., 2009) for C. In all these specification languages, the expected program properties are expressed by annotating the program with some kind of first-order formulas over program states. Those annotations are typically classified as preconditions and postconditions on subprograms (these formulas together forming a subprogram *contract*), invariants on loops, assertions at particular program points, data invariants, *etc.*

Verifying that a given program meets its given specification amounts to generating *verification conditions* (VC), which are first-order logic formulas whose validity must be checked by a theorem prover. A central issue of this general approach is that whenever a proof attempt fails, it does not necessarily mean that the expected property is wrong: it might be because the program needs additional annotations. A classical example of this is the necessity of invariants on loops, as in the following very simple code:

```
1  i := 0;
2  while i < 10 loop i := i + 1; endloop;
3  assert i = 10;
```

Proving the assertion on line 3 in Hoare's logic (or using Dijkstra's weakest precondition calculus) typically requires annotating the program with loop invariant $i \leq 10$. Otherwise, the only information at loop exit is the negation of the loop test *not* $(i < 10)$. Together with the loop invariant, the negation of the loop test ensures that $i = 10$.

Deductive verification is called *modular* when a subprogram can be verified independently of its callers and its callees. Typically, a manually defined precondition summarizes the constraints from the subprogram call-sites; calls inside the subprogram are handled by replacing them with the contract of those subprograms called. In that context, similarly to what we said about missing loop invariants, proof failure may happen if preconditions or postconditions are not precise enough. Deductive verification of programs thus typically requires one to manually annotate programs with detailed contracts, which prevents this approach from scaling up to programs of large size. To make the approach practical, it is mandatory to propose methods to automatically infer annotations.

There have been a lot of different techniques proposed to generate invariants. Probably the most common approach is obtained by applying classical forward abstract interpretation on a whole program, and handling subprograms calls by propagating information through the corresponding body. Such an approach is not modular since it requires the whole program, and it cannot always generate preconditions that are precise enough since it proceeds forward only. On the contrary, the approach we propose in this paper proceeds modularly, and it is able to generate both precise pre- and postconditions for safety checking by combining forward and backward analyses. Notice that our approach summarizes each function based only on the summaries for the functions it calls. For example, our approach will not be able to exploit the fact that a call to F is always followed by a call to G in the generation of summaries for both F and G, but only in their callers. We will extensively compare our approach with existing works in Section 6. Our contributions are summarized as follows:

- Section 2 is devoted to preliminaries. We introduce a core language for programs which manipulate integers and arrays, which is not classical since control statements contain exceptions. We recall the basics of abstract interpretation and weakest precondition calculus on this language, with an emphasis on the precise treatment of array cells. An originality of our approach is that abstract interpretation is *intraprocedural*, which means that a subprogram call is handled by interpreting a given postcondition and not by expanding the code.
- In Section 3, we present our technique for generating pre- and postconditions of subprograms. In that section we consider only alias-free programs, as defined in Section 2.
- In Section 4, we extend our approach to programs with possible aliasing. The generation of annotations is guided by an analysis of pointer separation to deal with aliasing.
- Our technique is implemented within the Frama-C environment for static analysis of C programs (http://frama-c.cea.fr). We report in Section 5 on experiments made on a library and two benchmarks of vulnerabilities of C code.

Finally, please note that most of the results presented here appeared in the PhD of Moy (2009), where many more technical details can be found if needed.

$$
\begin{aligned}
op &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{\&\&} \mid \texttt{||} \\
uop &::= \texttt{-} \mid \texttt{not} \\
e &::= \texttt{true} \mid \texttt{false} \mid n \mid x \mid e\ op\ e \mid uop\ e \mid x[e] \mid (e) \\
i &::= x\texttt{:=}e\texttt{;} \mid x[e]\texttt{:=}e\texttt{;} \mid \texttt{assert}\ e\texttt{;} \mid x\texttt{:=}f(e, .., e)\texttt{;} \mid x := \texttt{new}\ t[e]\texttt{;} \\
s &::= i \mid s\ s \mid \texttt{return}\ e\texttt{;} \mid \texttt{throw}\ Exc(e)\texttt{;} \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{if}\ e\ \texttt{then}\ s \mid \texttt{loop}\ s\ \texttt{endloop;} \\
&\quad \mid \texttt{try}\ s\ \texttt{catch}\ (Exc\ x)\ s \\
d &::= \texttt{exception}\ t\ Exc \mid t\ x \mid t\ f(t\ x, .., t\ x)\{s\} \\
t &::= integer \mid bool \mid void \mid t[]
\end{aligned}
$$

Fig. 1. Grammar of our core language.

2. Preliminary material

In this section, we introduce the core programming language on which we perform analyses. This language defines values of integer type and array type, but no value of pointer type, so that aliasing is not a concern for now. In Section 4, we will return to this definition and add the possibility of aliasing with values of pointer type. On this language, we describe our basic settings for abstraction interpretation, weakest precondition calculus, and quantifier elimination.

To illustrate our approach we consider the following running example.

Example 1. Linear search is a small function, written as follows in the C programming language:

```
1  int linear_search(long t[], int len, long key) {
2    int i;
3    for (i=0; i < len; i++) {
4        if (t[i] == key) return i;
5      }
6      return -1;
7  }
```

It searches for the value *key* in array *t* between indexes 0 and $len - 1$. It returns the index if it is found, or -1 otherwise.

2.1. The core language

The core language that we consider is given by the grammar of Fig. 1. *e* is the non-terminal for expressions, which cannot produce any side effects. The usual rules of precedence apply, so that `x + y == z` parses as `(x + y) == z`. We allow chains of comparisons to denote a conjunction of the equivalent individual comparisons, so that `x < y && y <= z` can be written more succinctly as `x < y <= z`. *i* is the non-terminal for instructions, *i.e.*, basic statements which cannot modify the control flow of execution. *s* is the non-terminal for statements, and *d* stands for declarations of exceptions and subprograms. Note that we do not allow global variables in our core language, for simplicity. *v* denotes variable identifiers and *Exc* denotes exception names. The basic types are booleans, unbounded integers, the void type to simulate subprograms with no result and unbounded arrays.

This core language is quite non-standard in its basic types: unbounded integers are rarely found in real programming languages, not to mention unbounded arrays. These types abstract over the usual machine integers and bounded arrays found in real programming languages. The usual constraints over integers and arrays to prevent overflows will be translated in our intermediate language as explicit assertions (see Section 2.2 below), so that we do not have to worry about undefined expressions in our analyses. All parameters are passed by copy, including parameters of array type. Operator `new` returns an array for now, not a pointer to an array like in C++, so that aliasing is not possible (this will change in Section 4).

The core language is also quite non-standard in its basic constructions of statements, although it is quite close to the intermediate language of ESC/Java2 (Leino et al., 1999): it natively supports exception throwing and catching. It is indeed inspired from the Jessie intermediate language (Marché, 2007) of the Why platform (Filliâtre and Marché, 2007). First, it allows us to encode other complex control flow statements like `break` or `continue` statements of C and Java, and Java exceptions. Second, using exceptions as core statements will allow us later to describe both abstract interpretation techniques and weakest precondition calculi as deduction rules over symbolic judgements, instead of computation rules over control flow graphs.

Example 2. The following is our running example of linear search, expressed in our core language:

```
1  exception void Break;
2
3  integer linear_search(integer[] t, integer len, integer key) {
4    integer i;
5    i := 0;
6    try
7      loop
8        if not (i < len) then throw Break ();
9        if t[i] == key then return i;
10       i := i + 1;
11     endloop;
12   catch (Break v) return −1;
13 }
```

Notice the use of an infinite loop with an exception to simulate the `for` loop.

2.2. Safety checking as assertion checking

Basic safety properties of all programs are the absence of division by zero, the absence of integer overflow, and the absence of memory access error. In our core language, we reduce safety checking to assertion checking, by expressing suitable assertions on unbounded integers that originate either from program variables or from our memory model instrumentation of the program.

Example 3. Here is our running example now equipped with assertions for safety checking.

```
1  requires INT_MIN <= len <= INT_MAX;
2  requires LONG_MIN <= key <= LONG_MAX;
3  integer linear_search(integer[] t, integer len, integer key) {
4    integer i;
5    i := 0;
6    try
7      loop
8        if not (i < len) then throw Break ();
9        assert (0 <= i < array_length(t));
10       if t[i] == key then return i;
11       assert (INT_MIN <= i+1 <= INT_MAX);
12       i := i + 1;
13     endloop;
14   catch (Break v) return −1;
15 }
```

In the first assertion on line 9, `array_length` allows us to express safety of memory dereferencing, using the annotation-only function `array_length` which provides information about the length of array t.

In the second assertion on line 11, `INT_MIN` and `INT_MAX` denote the bounds for machine integers of type `int`, typically -2^{31} and $2^{31} - 1$ for 32-bits signed integers. Notice also the additional preconditions introduced by `requires` which correspond to type information for parameters.

2.3. Forward abstract interpretation

Abstract interpretation (Cousot and Cousot, 1979) is a theory of abstract semantics of programs. Values taken by the variables during the execution of a program are mapped into an *abstract domain*. Such a domain is built upon an internal *abstract lattice*, an algebraic structure that describes the ordering relation between elements in the model, so that each iteration on the program's control flow can only increase abstract values w.r.t. this ordering.

2.3.1. Abstract lattices

A lattice is usually best described by a tuple $(L, \sqsubset, \bot, \top, \sqcup, \sqcap)$, where:

- L is the set of elements in the lattice;
- $\sqsubset$ is the ordering relation;
- $\bot$ is the least element in the lattice;
- $\top$ is the greatest element in the lattice;
- $\sqcup$ is the union (join) of elements in the lattice;
- $\sqcap$ is the intersection (meet) of elements in the lattice.

An important property is that $\sqcup$ and $\sqcap$ are consistent with respect to the ordering $\sqsubset$, which amounts to

$$x \sqsubset y \Rightarrow (x \sqcup y = y \wedge x \sqcap y = x).$$

Connection between the program and its model is done through a Galois connection (α, γ), where the abstraction function α maps each set of concrete program states into an abstract element of L, and the concretization function γ maps each abstract element of L to a set of concrete program states. For the lattice to correctly over-approximate the set of possible program behaviors, the following properties are required:

$$\gamma(x \sqcup y) \supset \gamma(x) \cup \gamma(y), \tag{1}$$

$$\gamma(x \sqcap y) \subset \gamma(x) \cap \gamma(y), \tag{2}$$

$$\gamma \circ \alpha(s) \supset s. \tag{3}$$

Most lattices we use in practice are convex, or stable by intersection, which means that we can rewrite Eq. (2) into

$$\gamma(x \sqcap y) = \gamma(x) \cap \gamma(y). \tag{4}$$

2.3.2. Transfer functions

Given an abstract lattice L, an abstract domain D can be built upon L by defining additional *transfer functions* that over-approximate the effect of a program statement on a value of L:

- D.`test` is the transfer function for test;
- D.`assign` is the transfer function for assignment;
- D.`forget` is the transfer function for reset.

The effect of an assignment is over-approximated either by D.`assign` when the right-hand side of the assignment is simple enough, or by D.`forget` otherwise, in which case all information about the value of the variable assigned is lost. These transfer functions are usually built upon the underlying lattice operations. The essential property of these transfer functions is monotonicity, which amounts to

$$x_1 \sqsubset x_2 \Rightarrow D.f(x_1) \sqsubset D.f(x_2),$$

for $D.f$ any of D.`test`, D.`assign` or D.`forget`, and all other parameters of these operators (not mentioned here) being equal.

Other operations of the abstract domain are simple wrappers on underlying lattice operations:

- D.`union` is the join operation $\sqcup$;
- D.`included` is the ordering relation $\sqsubset$.

D.`union` is used to perform unions of abstract values at junctions of paths during the propagation leading to the definition of the abstract model of a program. D.`included` is used to check the validity of a proposition expressed as an element of L.

There is generally a last operation defining an abstract domain:

- D.`widen` is the widening operation.

When the lattice L has bounded height, convergence of the propagation is ensured by the monotonicity of transfer functions. When L has infinite height, convergence is not ensured by monotonicity. D.`widen` allows us to ensure or to accelerate convergence. It is used to directly jump to an over-approximation of the set of reachable states of the program. Formally speaking, D.`widen` must guarantee that given any ascending chain of abstract values abs_i, the chain abs'_i defined by

$$\begin{aligned} abs'_0 &= abs_0 \\ abs'_{i+1} &= D.\texttt{widen}(abs'_i, abs_i) \end{aligned}$$

converges in finite time, i.e.

$$\exists\, k, \forall i, k \leq i \rightarrow abs'_{i+1} = abs'_i.$$

2.3.3. Abstract domains in practice

Abstract domains that are useful for us in practice are either:

- non-relational domains, which bound or restrict in some way the value of individual variables: sign (abbreviated as **Sign** below), interval (**Interv**), congruence, etc.
- relational domains, which provide relations between the value of two or more variables. Domains are sets of relation formulas, and are classified by the supported form of formulas:
  - Difference Bound Matrices (Dill, 1989): $a \leq x - y \leq b$ s.t. $a, b \in \mathbb{Z} \cup \pm\infty$ and x, y variables.
  - Octagons (**Oct**) (Miné, 2006): $a \leq x \pm y \leq b$ s.t. $a, b \in \mathbb{Z} \cup \pm\infty$ and x, y variables.
  - Linear equalities (Karr, 1976): $\sum a_i x_i = b$ s.t. $a_i, b \in \mathbb{Z}$ and x_i variables.
  - Polyhedrons (**Poly**) (Cousot and Halbwachs, 1978; Colon et al., 2003): $\sum a_i x_i \leq b$ s.t. $a_i, b \in \mathbb{Z}$ and x_i variables.

The most complex abstract domains still commonly used are relational domains. They vary in complexity, from quadratic or cubic complexity for weakly relational domains which relate the value of 2 or 3 variables, to exponential complexity in the worst case for full relational domains which relate the value of an unbounded number of variables.

2.3.4. Intraprocedural abstract interpretation

The intraprocedural abstract interpretation of programs is a data-flow analysis on instructions and statements. The effect of an instruction s is described by judgment

$$A_1 \vdash s \longrightarrow A_2$$

with the meaning that executing instruction s from a state in any concretization of abstract state A_1 leads to some state in the concretization of the abstract state A_2. We consider an abstract domain over both integer variables in the subprogram, like `y`, and integer array cells in the subprogram, like `x[y]`, where `x` is an array variable and `y` is an integer variable. We present a slightly non-standard set of rules for abstract interpretation over instructions, such that more than one rule may apply. In that case, all rules that apply should be taken into account in turn. This allows us to treat array cells like `x[y]` as variables in abstract interpretation. Notice that we only track values of expressions that syntactically appear in the program text.

In these rules, we mention a function *equal* on array indexes, which returns *true* if we know that its parameters are equal at the program point of interest, *false* if we know that its parameters are not equal at the program point of interest, and *dont_know* otherwise. Of course, the precision of our analysis depends upon the precision of this function *equal*, with a default function always returning *dont_know* unless its parameters are syntactically equal.

$$(\textsc{assign-var})\frac{A(e) = v}{A \vdash y := e \longrightarrow D.\texttt{assign}(A, y, v)}$$

$$(\textsc{ignore-var})\frac{A(e) \text{ undefined}}{A \vdash y := e \longrightarrow D.\texttt{forget}(A, y)}$$

$$(\textsc{assign-index})\frac{}{A \vdash y := e \longrightarrow D.\texttt{forget}(A, x[y])}$$

$$(\textsc{assign-array-equal})\frac{equal(e_1, y) = true \wedge A(e_2) = v}{A \vdash x[e_1] := e_2 \longrightarrow D.\texttt{assign}(A, x[y], v)}$$

$$(\textsc{assign-array})\frac{equal(e_1, y) = dont\text{-}know \wedge A(e_2) = v}{A \vdash x[e_1] := e_2 \longrightarrow D.\texttt{union}(A, D.\texttt{assign}(A, x[y], v))}$$

$$(\textsc{ignore-array-equal})\frac{equal(e_1, y) = true \wedge A(e_2) \text{ undefined}}{A \vdash x[e_1] := e_2 \longrightarrow D.\texttt{forget}(A, x[y])}$$

$$(\textsc{ignore-array})\frac{equal(e_1, y) = dont\text{-}know \wedge A(e_2) \text{ undefined}}{A \vdash x[e_1] := e_2 \longrightarrow D.\texttt{forget}(A, x[y])}$$

$$(\textsc{ignore-assert})\frac{}{A \vdash \texttt{assert}\, e \longrightarrow A}$$

$$(\textsc{ignore-alloc})\frac{}{A \vdash x := \texttt{new}\, t[e] \longrightarrow D.\texttt{forget}(A, x[y])}$$

$$(\textsc{call})\frac{}{A \vdash x := f(e_1, \ldots, e_k) \longrightarrow D.\texttt{test}(A, post_f[x_i \mapsto e_i, \mathsf{result} \mapsto x])}$$

Fig. 2. Abstract interpretation of instructions.

Fig. 2 presents the abstract interpretation of instructions. For an expression e, $A(e)$ denotes the abstract value it has in A (undefined if e is not supported by domain D, which is typically the case for non-linear arithmetic operations). Assigning to an integer variable updates the value associated to this variable in rule (ASSIGN-VAR), and it forgets the value associated to all array cells based on this variable as index in rule (ASSIGN-INDEX). Assigning to an array cell has an effect on all variables representing cells in this array, which may correspond to the same index, as described in rules (ASSIGN-ARRAY-EQUAL) and (ASSIGN-ARRAY). In rule (CALL), the postcondition $post_f$ must denote the postcondition of subprogram f, in which the result keyword denotes the returned value.

The effect of a statement s is described by judgment

$$A \vdash s \longrightarrow (A^N, X_i \Rightarrow A^{X_i})$$

with the meaning that executing statement s from a state in the concretization of abstract state A leads to a state in the concretization of one of the abstract states depending on the outcome of s:

- A^N if the outcome is normal termination;
- A^{X_i} if the outcome is that exception X_i is thrown.

Note that for uniformity we consider `return e;` as if it was a special exception `throw (Return e);` which is implicitly caught at subprogram's exit.

Fig. 3 presents the abstract interpretation of statements. Rule (LOOP-UNROLL) and (LOOP-WIDEN) are the only recursive rules. They implicitly suggest an iterative procedure to discover a fix-point to the data-flow propagation, namely to iterate the propagation through the loop body until it converges,

$$(\textsc{instr})\frac{A \vdash i \longrightarrow A'}{A \vdash i \longrightarrow (A', \varepsilon)}$$

$$(\textsc{seq})\frac{A_1 \vdash s_1 \longrightarrow (A_1^N, X_i \Rightarrow A_1^{X_i}) \quad A_1^N \vdash s_2 \longrightarrow (A_2^N, X_i \Rightarrow A_2^{X_i})}{A_1 \vdash s_1; s_2 \longrightarrow (A_2^N, X_i \Rightarrow D.\texttt{union}(A_1^{X_i}, A_2^{X_i}))}$$

$$(\textsc{return})\frac{A(e) = v}{A \vdash \texttt{return}\, e \longrightarrow (\bot, \mathit{Return} \Rightarrow D.\texttt{assign}(A, \mathsf{result}, v))}$$

$$(\textsc{return-ignore})\frac{A(e) \text{ undefined}}{A \vdash \texttt{return}\, e \longrightarrow (\bot, \mathit{Return} \Rightarrow A)}$$

$$(\textsc{throw})\frac{A(e) = v}{A \vdash \texttt{throw}\ (X\ e) \longrightarrow (\bot, X \Rightarrow D.\texttt{assign}(A, \mathsf{result}, v))}$$

$$(\textsc{throw-ignore})\frac{A(e) \text{ undefined}}{A \vdash \texttt{throw}\ (X\ e) \longrightarrow (\bot, X \Rightarrow A)}$$

$$(\textsc{if})\frac{D.\texttt{test}(A, e) \vdash s_1 \longrightarrow (A_1^N, X_i \Rightarrow A_1^{X_i}) \quad D.\texttt{test}(A, \neg e) \vdash s_2 \longrightarrow (A_2^N, X_i \Rightarrow A_2^{X_i})}{A \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \longrightarrow (D.\texttt{union}(A_1^N, A_2^N, X_i \Rightarrow D.\texttt{union}(A_1^{X_i}, A_2^{X_i})))}$$

$$(\textsc{loop-unroll})\frac{A \vdash s \longrightarrow (A_1^N, X_i \Rightarrow A_1^{X_i}) \quad A_1^N \vdash \texttt{loop}\, s\, \texttt{endloop} \longrightarrow (A_2^N, X_i \Rightarrow A_2^{X_i})}{A \vdash \texttt{loop}\, s\, \texttt{endloop} \longrightarrow (D.\texttt{union}(A_1^N, A_2^N), X_i \Rightarrow D.\texttt{union}(A_1^{X_i}, A_2^{X_i}))}$$

$$(\textsc{loop-widen})\frac{A \vdash s \longrightarrow (A_1^N, X_i \Rightarrow A_1^{X_i}) \quad A_1^N \vdash \texttt{loop}\, s\, \texttt{endloop} \longrightarrow (A_2^N, X_i \Rightarrow A_2^{X_i})}{A \vdash \texttt{loop}\, s\, \texttt{endloop} \longrightarrow (D.\texttt{widen}(A_1^N, A_2^N), X_i \Rightarrow D.\texttt{widen}(A_1^{X_i}, A_2^{X_i}))}$$

$$(\textsc{loop-converge})\frac{A \vdash s \longrightarrow (A_1^N, X_i \Rightarrow A_1^{X_i}) \quad D.\texttt{included}(A_1^N, A)}{A \vdash \texttt{loop}\, s\, \texttt{endloop} \longrightarrow (A, X_i \Rightarrow A_1^{X_i})}$$

$$(\textsc{try})\frac{A \vdash s_1 \longrightarrow (A_1^N, X_i \Rightarrow A_1^{X_i}, X \Rightarrow A_1^X) \quad A_1^X[\mathsf{result} \mapsto v] \vdash s_2 \longrightarrow (A_2^n, X_i \Rightarrow A_2^{X_i})}{A \vdash \texttt{try}\ \ s_1\ \ \texttt{catch}\ \ (X\ v)\, s_2 \longrightarrow (D.\texttt{union}(A_1^N, A_2^N, X_i \Rightarrow D.\texttt{union}(A_1^{X_i}, A_2^{X_i})))}$$

Fig. 3. Abstract interpretation of statements.

which is expressed by rule (LOOP-CONVERGE). When convergence is not ensured, it is the role of the widening function to provide it. The starting point for the intraprocedural abstract interpretation is the value $\top$ at the entry point of the function, and $\bot$ everywhere else.

Taking annotations (e.g. preconditions) into account simply consists in calling $D.\texttt{test}$ to further constrain the current abstract value. Checking that a proposition p holds is possible if p can be expressed as an abstract value p_{abs}: then, p holds if $D.\texttt{included}(A, p_{\text{abs}})$, where A is the current abstract state.

2.3.5. Illustration on linear search

If we run our intraprocedural abstract interpretation in the unannotated version of linear search (Example 3) with various domains, we get the results presented on Fig. 4. It does not allow us to prove integer safety for function linear_search which amounts to check assertion:

$$\mathit{INT\_MIN} \leq i + 1 \leq \mathit{INT\_MAX}.$$

	loop invariant	invariant at assert	postcondition
Sign	$0 \leq i$	$0 \leq i$	
Interv	$0 \leq i$	$0 \leq i$	$-1 \leq \mathsf{result}$
Oct & Poly	$0 \leq i$	$0 \leq i < \mathit{len}$	$-1 \leq \mathsf{result}$

Fig. 4. Abstract interpretation of unannotated function linear_search.

$$
\begin{aligned}
W(y := e, P) &= P[y \mapsto e; x[y] \mapsto ?] \\
W(x[e_1] := e_2, P) &= P[x[y] \mapsto (\mathit{if}\ \mathit{equal}(y, e_1) = \mathit{true}\ \mathit{then}\ e_2 \\
&\qquad\qquad \mathit{elsif}\ \mathit{equal}(y, e_1) = \mathit{dont\text{-}know}\ \mathit{then}\ ? \\
&\qquad\qquad \mathit{else}\ x[y])] \\
W(\mathtt{assert}\ e, P) &= e \wedge P \\
W(x := f(\vec{e_i}), P) &= \mathit{pre}_f[v_i \mapsto e_i] \wedge \\
&\qquad (\mathit{post}_f[v_i \mapsto e_i, \mathsf{result} \mapsto r'] \Longrightarrow P[x \mapsto r']) \\
W(x := \mathtt{new}\ \ t[e], P) &= P[\mathit{array\text{-}length}(x) \mapsto e]
\end{aligned}
$$

Fig. 5. Weakest preconditions of instructions.

Taking into account preconditions from parameter types (Example 3) to strengthen invariants does not allow us to prove the assertion either.

2.4. Weakest precondition calculus

A Hoare triple, denoted as $\{P\}s\{Q\}$ is made of a precondition P, a statement s and a postcondition Q. The partial correctness of s is defined as the validity of the triple, that is if s is executed in any state satisfying P, then when it terminates, the halting state satisfies Q. Weakest precondition calculus (Dijkstra, 1976) is a way to automate this process: $W(s, Q)$ is a formula which is the weakest to request on the initial state of s to guarantee the postcondition Q. Thus, checking a Hoare triple reduces to checking $P \Rightarrow W(s, Q)$ with some theorem prover.

The rules for computing the weakest precondition for instructions of our core language are given in Fig. 5. The notation $[x \mapsto ?]$ means that the variable x is replaced by a fresh name in the formula. This is the case when assigning to an integer variable, which causes all array cells using this variable as index to be replaced by fresh variables. This is the case when assigning to an array cell, which causes all possibly equal (but not for sure) array cells to be replaced by fresh variables.

In our core language with exceptions, a statement may have a normal termination outcome but also may end in an exceptional state. Thus Hoare triples must contain exceptional postconditions (Filliâtre, 2003; Burdy et al., 2004): $\{P\}s\{Q \mid X_i \Rightarrow Q_i\}$ means that whenever s ends in exception X_i then Q_i is satisfied. The weakest precondition is then naturally defined under the form $W(s, Q, X_i \Rightarrow Q_i)$. Classically, result is renamed into a fresh r'.

The rules for computing the weakest precondition for statements are given in Fig. 6. Whenever a variable y gets quantified, all array variables based on y, *e.g.*, $x[y]$, get quantified too.

Example 4. On the linear search code of Example 2, let us check that the postcondition $Q \equiv \mathsf{result} \geq 0 \Longrightarrow t[\mathsf{result}] = \mathit{key}$ is valid. We need to compute $W(b, \mathtt{true}, \mathtt{Return} \Rightarrow Q)$, where b is the body of the search function.

- the weakest precondition of Q through `return -1` is $-1 \geq 0 \Longrightarrow \ldots$ which is true.
- Applying the rule for `try ... catch .,.`, we need to compute $W(l, \mathtt{true}, \mathtt{Return} \Rightarrow Q, \mathit{Break} \Rightarrow \mathtt{true})$ where l is the infinite loop.
- Applying the rule for loops, with `true` as loop invariant, we have to compute $W(lb, \mathtt{true}, \mathtt{Return} \Rightarrow Q, \mathit{Break} \Rightarrow \mathtt{true})[i \mapsto i']$ where lb is the loop body.

$$
\begin{aligned}
&W(s_1; s_2, Q, X_i \Rightarrow Q_i) = W(s_1, W(s_2, Q, X_i \Rightarrow Q_i), X_i \Rightarrow Q_i)\\
&W(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q, X_i \Rightarrow Q_i) =\\
&\qquad (e \Longrightarrow W(s_1, Q, X_i \Rightarrow Q_i)) \wedge (\neg e \Longrightarrow W(s_2, Q, X_i \Rightarrow Q_i))\\
&W(\texttt{loop } \{ \texttt{ invariant } I \,\}\, s \texttt{ endloop}, Q, X_i \Rightarrow Q_i) =\\
&\qquad I \wedge ((I \Longrightarrow W(s, I, X_i \Rightarrow Q_i))[x_j \mapsto y_j])\\
&W(\texttt{return } e, Q, \mathit{Return} \Rightarrow Q_R, X_i \Rightarrow Q_i) = Q_R[\mathsf{result} \mapsto e]\\
&W(\texttt{throw } (Xe), Q, X \Rightarrow Q_X, X_i \Rightarrow Q) = Q_X[\mathsf{result} \mapsto e]\\
&W(\texttt{try } s_1 \texttt{ catch } (Xv)s_2, Q, X_i \Rightarrow Q_i) =\\
&\qquad W(s_1, Q, X \Rightarrow W(s_2, Q, X_i \Rightarrow Q_i), X_i \Rightarrow Q_i)
\end{aligned}
$$

Fig. 6. Weakest preconditions of statements.

$$
\begin{aligned}
W^+(y := e, P) &= W(y := e, P)\\
W^+(x[e_1] := e_2, P) &= W(x[e_1] := e_2, P)\\
W^+(\texttt{assert } e, P) &= \mathit{if\ target\ then}\ e \wedge P\ \mathit{else}\ P\\
W^+(x := f(\vec{e_i}), P) &= (\mathit{if\ target\ then}\ \mathit{pre}_f[v_i \mapsto e_i]\ \mathit{else\ true}) \wedge\\
&\qquad (\mathit{post}_f[v_i \mapsto e_i, \mathsf{result} \mapsto r'] \Longrightarrow P[x \mapsto r'])\\
W^+(x := \texttt{new } t[e], P) &= W(x := \texttt{new } t[e], P)
\end{aligned}
$$

Fig. 7. Specialized weakest preconditions of instructions.

- After applying rules for sequence and if-then-else, it reduces to

$$
\begin{aligned}
t[i] = \mathit{key} &\Longrightarrow W(\texttt{return } i, \texttt{true}, \texttt{Return} \Rightarrow Q, \mathit{Break} \Rightarrow \texttt{true})\\
\mathit{not}(i < \mathit{len}) &\Longrightarrow W(\texttt{throw } \mathit{Break}, \texttt{true}, \texttt{Return} \Rightarrow Q, \mathit{Break} \Rightarrow \texttt{true})
\end{aligned}
$$

  which further reduces to

$$
\begin{aligned}
t[i] = \mathit{key} &\Longrightarrow Q[\mathsf{result} \mapsto i]\\
\mathit{not}(i < \mathit{len}) &\Longrightarrow \texttt{true}
\end{aligned}
$$

  which are both tautologies.

Similarly, it is possible to establish the stronger postcondition $\mathsf{result} = -1 \Longrightarrow \forall k, 0 \le k < \mathit{len} \Longrightarrow t[k] \neq \mathit{key}$ by adding the loop invariant $\forall k, 0 \le k < i \Longrightarrow t[k] \neq \mathit{key}$.

Example 5. To show how we handle array cells, let us compute the weakest precondition of various formulas Q through the simple assignment `t[i] := t[j];`.

- For $Q \equiv t[i] = 0$, we simply get $t[j] = 0$.
- For $Q \equiv t[k] = 0$, it depends upon the result of $\mathit{equal}(i, k)$, which represents our knowledge at this point of the possible equality between i and k.
  . If $\mathit{equal}(i, k) = \mathit{true}$, we get $t[j] = 0$.
  . If $\mathit{equal}(i, k) = \mathit{false}$, we get $t[k] = 0$.
  . If $\mathit{equal}(i, k) = \mathit{dont\_know}$, we get $v = 0$ for a fresh variable v.

For each assertion to check in the code, which can be either a plain assertion or the precondition of a function called, we specialize function W into function W^+ which computes the precondition uniquely associated to this assertion. Rules for W^+ are almost the same as those for W, except that paths which do not lead to the targeted assertion are ignored. In the rules for instructions presented in Fig. 7, the treatment of assertions and function calls is different for the targeted assertion (when condition *target* is true) and other assertions. In the rules for statements presented in Fig. 8, the treatment of loops does not require us to prove the loop invariant I.

$$
\begin{aligned}
&W^+(s_1; s_2, Q, X_i \Rightarrow Q_i) = W(s_1; s_2, Q, X_i \Rightarrow Q_i)\\
&W^+(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q, X_i \Rightarrow Q_i) =\\
&\qquad W(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, Q, X_i \Rightarrow Q_i)\\
&W^+(\texttt{loop } \{ \texttt{ invariant } I \}\, s \texttt{ endloop}, Q, X_i \Rightarrow Q_i) =\\
&\qquad (I \Longrightarrow W(s, I, X_i \Rightarrow Q_i))[x_j \mapsto y_j]\\
&W^+(\texttt{return } e, Q, \mathit{Return} \Rightarrow Q_R, X_i \Rightarrow Q_i) =\\
&\qquad W(\texttt{return } e, Q, \mathit{Return} \Rightarrow Q_R, X_i \Rightarrow Q_i)\\
&W^+(\texttt{throw } (Xe), Q, X \Rightarrow Q_X, X_i \Rightarrow Q) = W(\texttt{throw } (Xe), Q, X \Rightarrow Q_X, X_i \Rightarrow Q)\\
&W^+(\texttt{try } s_1 \texttt{ catch } (Xv)s_2, Q, X_i \Rightarrow Q_i) = W(\texttt{try } s_1 \texttt{ catch } (Xv)s_2, Q, X_i \Rightarrow Q_i)
\end{aligned}
$$

Fig. 8. Specialized weakest preconditions of statements.

Given an assertion A, we call $W^+(\mathit{prog}, \mathit{true})$ the precondition of assertion A. This is the precondition we will be considering in the following.

2.5. Quantifier elimination

The last ingredient we need for our approach is quantifier elimination. It does not directly apply to analysis of programs, but it is a useful technique for simplifying the logic formulas we infer with other techniques.

Mainly, what we need is quantifier elimination for formulas in linear integer arithmetic (*i.e.*, Presburger arithmetic). Unfortunately, quantifier elimination for such formulas is inherently triply exponential (Weispfenning, 1997), which is far too expensive in practice, as we checked in our experiments with Cooper's method. Instead, we turn to quantifier elimination for rational (or real) linear arithmetic, for which there exists algorithms with a doubly exponential complexity (Bradley and Manna, 2007).

Classically, we exploit the particular form of the formulas generated by weakest preconditions. Our quantified formulas are universally quantified prenex formulas, meaning only universal quantifiers $\forall$ appear in front of a quantifier-free formula. Quantified variables correspond to all the fresh variables introduced during weakest preconditions (*e.g.*, for the loop rule or *dont_know* values) and local variables, that have meaning only inside the function, not at call-site. Then, quantifier elimination over the rationals only returns a stronger formula than quantifier elimination over the integers, which is a correct approximation in our case.

In practice, we rewrite the universal formula $\forall\ \vec{x_i}\ .\ P$ into the equivalent $\neg(\exists\ \vec{x_i}\ .\ \neg P)$ and we transform formula $\neg P$ into its disjunctive normal form (DNF), so that the existential quantifier distributes over all disjuncts and the Fourier–Motzkin method to eliminate quantifiers can be applied individually to each disjunct. The conversion to a DNF formula has exponential complexity and the Fourier–Motzkin method has doubly exponential complexity in theory, but closer to exponential in practice (Monniaux, 2008). Therefore, we obtain this way a doubly exponential complexity in practice, which is the best complexity reached by existing algorithms.

3. Inferring contracts of alias-free programs

We now consider the problem of inferring contracts modularly for alias-free programs, based on the techniques described in Section 2. Absence of aliasing is ensured in our case by the design of our core programming language : the base types are booleans, integers and arrays, which cannot be referenced by different names in a subprogram. This allowed us to define precise intraprocedural analyses by abstract interpretation and weakest precondition computation in Section 2.

3.1. Generic algorithm and its soundness

We suppose given a set of analyses:

- ForwAbsInt is a forward abstract interpretation as defined in Section 2.3.

- PreCond is a weakest precondition computation as defined in Section 2.4.
- QuantElim is a quantifier elimination procedure as defined in Section 2.5.

We now define a combination of these analyses that generates contracts for subprograms.

Algorithm 1. Algorithm InferGen:
input a program possibly annotated, and a subprogram P
output additional annotations for P

(1) Compute invariants I_p for each program point p of P by ForwAbsInt. Calls are handled using given annotations of other subprograms.
(2) Use the disjunction of I_p for each exit point p to strengthen P's postcondition.
(3) For each `assert` C in P at program point p:
  (a) let w_p be the formula $I_p \Longrightarrow C$ (C weakened by I_p),
  (b) let ϕ_p be the result of PreCond(w_p),
  (c) let ψ_p be the result of QuantElim(ϕ_p),
  (d) use ψ_p to strengthen P's precondition.

Notice that algorithm InferGen builds on the contracts of those subprograms that are called inside the body of the subprogram currently analyzed. This suggests that programs should be better handled from leaf subprograms to root subprograms, proceeding bottom-up in the call-graph.

Theorem 6. *If*

(1) ForwAbsInt *produces only correct invariants*
(2) PreCond(ϕ) *produces a sufficient condition for* ϕ
(3) QuantElim(ϕ) *produces an equivalent or stronger proposition than* ϕ

then the InferGen *algorithm can only produce sufficient preconditions and correct postconditions.*

Proof. Since ForwAbsInt produces only correct invariants, the generated postconditions are correct, and for any assert C at some program p, the generated invariant I_p holds at p. Since PreCond produces sufficient preconditions, and since QuantElim can only strengthen formulas, the generated preconditions guarantee that $I_p \Longrightarrow C$ holds at program point p. Thus by a simple *modus ponens*, C holds at p. □

3.2. Weak inference

The InferWeak method is an instance of InferGen as follows.

Algorithm 2. Algorithm InferWeak is the specialization of InferGen where

- ForwAbsInt is forward abstract interpretation as in Section 2.3.
- PreCond is weakest precondition as in Section 2.4.
- QuantElim is Fourier–Motzkin method.

Example 7. On our running example of linear search, starting from the code of Example 3, and using either **Oct** or **Poly** domains.

- As seen in Section 2.3.5, at loop entry the invariant computed by abstract interpretation is

$$I_L := 0 \leq i.$$

- At function exit the computed invariants are respectively

$$0 \leq \mathsf{result} < \mathit{len}$$

and

$$\mathsf{result} = -1$$

hence postcondition is strengthened to

$$0 \leq \mathsf{result} < \mathit{len} \vee \mathsf{result} = -1.$$

- For the first assertion, we generate

  $$w_1 := 0 \leq i < len \Longrightarrow 0 \leq i < \text{array\_length}(t)$$

  whose weakest precondition is

  $$\begin{aligned} \phi_1 := \quad & \forall i \,.\, i = 0 \Longrightarrow \\ & \forall i' \,.\, 0 \leq i' \wedge i' < len \Longrightarrow (0 \leq i' < len \Longrightarrow 0 \leq i' < \text{array\_length}(t)) \end{aligned}$$

  and eliminating quantifiers leads to

  $$\psi_1 := 0 < len \Longrightarrow len \leq \text{array\_length}(t).$$

- For the second assertion, we generate

  $$w_2 := 0 \leq i < len \Longrightarrow \text{INT\_MIN} \leq i + 1 \leq \text{INT\_MAX}$$

  whose weakest precondition is

  $$\begin{aligned} \phi_2 := \quad & \forall i \,.\, i = 0 \Longrightarrow \\ & \forall i' \,.\, 0 \leq i' \wedge i' < len \Longrightarrow (0 \leq i' < len \Longrightarrow \text{INT\_MIN} \leq i + 1 \leq \text{INT\_MAX}) \end{aligned}$$

  and eliminating quantifiers leads to

  $$\psi_2 := 0 < len \Longrightarrow (\text{INT\_MIN} \leq 0 \wedge len \leq \text{INT\_MAX} + 1).$$

- The generated precondition is thus

  $$0 < len \Longrightarrow (len \leq \text{array\_length}(t) \wedge \text{INT\_MIN} \leq 0 \wedge len \leq \text{INT\_MAX} + 1).$$

From the example above, one may think that propagating backward the formula $I_p \Longrightarrow C$ instead of propagating C itself is not useful. This is the case here because the weakest precondition also adds the loop invariant as an hypothesis, and it appears that I_p is identical to the loop invariant since there are no side effects in the loop. However adding I_p as hypothesis is useful in general: first, in general I_p is different from I_L, and second, as we will see below, we can instantiate our generic algorithm with simpler precondition calculi, which in particular might ignore the loop invariant.

3.3. Controlling complexity

Overall, the classical computation of weakest preconditions can be seen as performing a part of the conversion to DNF, which leads to a combined doubly exponential complexity in practice. This is too costly to be applicable to real subprograms, which we checked in our experiments in Section 5. Thus, we devised variants of InferGen that differ from InferWeak by the precondition method PreCond used:

- InferStrong is based on a precondition method that computes a stronger formula than plain weakest preconditions, by ignoring statements that do not interfere directly with the formula being propagated backwards. A statement interferes with a formula ϕ either by constraining (in a test) or modifying (in an assignment or a call) a variable that occurs in ϕ. This is similar to the heuristic back-propagation of Janota (2007).
- InferElim completely replaces the weakest preconditions propagation by universally quantifying the assertion over all local variables (which may result in a much stronger precondition). *E.g.*, for the first assertion in Example 7, this corresponds to quantifying over local variable i without computing the weakest precondition:

  $$\phi_1 := \forall i \,.\, 0 \leq i \wedge i < len \Longrightarrow (0 \leq i < len \Longrightarrow 0 \leq i < \text{array\_length}(t)).$$

Although InferStrong and InferElim have the same complexity as InferWeak, they generate in practice much simpler quantified formulas, which leads to practical quantifier elimination for programs of a few hundred lines of code (see Section 5).

Example 8. On our running example, InferStrong and InferElim perform as well as InferWeak, leading to the same sufficient precondition. This is not always the case, as InferStrong and InferElim may fail to take into account relations between variables that stem from statements that InferStrong

ignores or that were not caught by the invariant used in InferElim. Indeed, InferElim performs as well as InferWeak in those cases where the invariant obtained by abstract interpretation is so precise that a backward propagation by weakest preconditions is not needed. Likewise, InferStrong performs as well as InferWeak in those cases where either the invariant obtained by abstract interpretation is precise enough, like for InferElim, or the strong preconditions computation captures the missing relational information.

4. Inferring contracts of pointer programs

We now consider the problem of inferring contracts modularly in the presence of pointers. Although accessing values through an indirection does not present any difficulty for reasoning about programs, pointers also allow sharing, which is a source of possible interference between read and write accesses, or between write accesses, that make precise analysis of programs more challenging.

In this section, we change the semantics of our core language without modifying its grammar, still given in Fig. 1. Arrays are now implicitly accessed through pointers : operator `new` returns a pointer to an array instead of an array, and array access `x[y]` denotes the access to array index `y` under pointer `x`. With this change, assignment between pointers makes aliasing possible. As an example of the problems due to possible aliasing, a simple relational abstract interpretation as described in Section 3 could easily infer that the following function

```
1  void incr (integer[] x, integer[] y) {
2    x[0] = y[0] + 1;
3  }
```

ensured postcondition

$$x[0] = y[0] + 1,$$

when treating arrays by copy. Now that we treat arrays by reference through pointers, this postcondition is not always true, because some caller could pass in the same pointer for `x` and `y` parameters. In that case, we would have instead the postcondition

$$x[0] = y[0].$$

In a modular analysis, we cannot analyze the set of calling contexts for a function in order to compute an overapproximation of the possible aliasing. We cannot either rely on the user to insert special annotations to describe possible aliasing relations, or forbid some aliasing relations, as it would not be practical and cost-effective on large programs with complex aliasing patterns. Our solution to this problem is thus a mix of an alias analysis and an alias control technique, that is incomplete but allows analyzing most programs that arise in practice.

4.1. Inferring regions: a type-based approach

Many analyses on pointer programs for type-safe languages (or restricted to type-safe programs in an otherwise unsafe language like C or C++) simply partition memory into disjoint regions, one region for each type or record field. At a logical level, each of these regions represents an array of all the memory locations where a value of this type may be found. Of course, these logical arrays do not correspond in general to contiguous physical memory locations. Then, a pointer is simply an index into a global array, and accessing a pointer is no more complex than indexing into an array. This partitioning based on types is the basis of our solution.

4.1.1. Steensgaard's region inference

Instead of partitioning memory according to types, Steensgaard's alias analysis partitions memory according to aliasing (Steensgaard, 1996). It computes a set of global regions that represent sets of disjoint memory locations, such that each pointer in the program is associated to a single region. Steensgaard's analysis is based on unification, like type inference, and it requires that all the program is available. It is a global flow-insensitive analysis, running in almost linear time.

We denote by region an equivalence class of pointers such that two pointers aliased necessarily belong to the same region. Initially, all pointers are assumed to belong to a different region. Then, regions are unified as necessary, based on a few rules:

- *assignment* - regions for pointers on both sides of an assignment are unified;
- *function call* - regions for pointers in corresponding parameters and arguments of a call are unified;
- *return* - regions for pointers in a function result and a term returned are unified.

Multiple levels of pointers in the program are reflected on the corresponding regions. Given a pointer in region ρ_1, whose dereference yields a pointer in region ρ_2, a special structure keeps track of the relation ρ_1 *points to* ρ_2. Then, unification of two regions also unifies the underlying regions pointed-to, and the regions pointed by the regions pointed-to, *etc.* As a result of this analysis, pointers that could be aliasing in some execution of the program necessarily belong to the same region.

As an example, for the program below:

```
1  void assign_first(integer[] u, integer[] v) {
2    u[0] := 0;
3  }
4
5  void main() {
6    integer[] x := new integer(1); integer[] y := new integer(1);
7    assign_first(x,y);
8  }
```

Steensgaard's analysis computes two regions ρ_1 = { x, u } and ρ_2 = { y, v }.

As a result, we know when analysing `assign_first` that `v[0]` is not modified by the assignment to `u[0]`, because `u` and `v` belong to two different regions.

4.1.2. A Contextual variant of Steensgaard's region inference

There are two issues with Steensgaard's regions: (i) lack of context sensitivity, which aliases more pointers than necessary, and (ii) lack of modularity, as the complete program is needed. Issue (i) arises on `assign_first` with the following code for `main`:

```
1  void assign_first(integer[] u, integer[] v) {
2    u[0] := 0;
3  }
4
5  void main() {
6    integer[] x := new integer(1); integer[] y := new integer(1);
7    integer[] z := new integer(1);
8    assign_first(x,y); assign_first(z,x);
9  }
```

In this program, `x` is used both in position of first and second argument to `assign_first`. Then, there is only one region computed by Steensgaard's analysis that groups all the pointers in this program: ρ_1 = { x, y, z, u, v }. As a result, we do not know whether `v[0]` is not modified by the assignment to `u[0]` in `assign_first`.

Hubert and Marché (2007) have described a solution to this problem, which is a contextual variant of Steensgaard's region inference. We call it contextual to distinguish it from the usual context-sensitive analysis: in a context-sensitive analysis, different mappings from pointers to regions would be computed for a given function, so that the best mapping is used for a given call ; in the contextual analysis of Hubert and Marché (2007), a single parametric mapping from pointers to regions is computed for a given function, so that its parameters are instantiated for a given call. This algorithm follows from an original idea from Talpin and Jouvelot (1991, 1992) for computing effects, used later on by Tofte and Talpin (1997) for static memory allocation.

Instead of computing global regions, like in Steensgaard's analysis, the contextual analysis computes parametric regions. A parametric region can be viewed as an additional parameter of the function, so that calls to this function must pass in one of their own parametric regions in scope.

This allows various calls to the same function to specify different region instances for each call, thus preventing aliasing due to merging of contexts. The analysis proceeds bottom-up from the leaves of the call-graph, one strongly connected component (SCC) at a time. While processing an SCC, the usual Steensgaard's analysis is performed, which unifies regions as required. When leaving an SCC, all the regions are labelled as parametric regions.

On the problematic version of our program above that leads Steensgaard's analysis to unify all regions together, the contextual analysis computes two parametric regions in function `assign_first`, namely regions `u` and `v`, and three local regions in function `main`, namely regions `x`, `y` and `z`. So instead of merging all regions in this example, it keeps all of them separated! As a result, we know that `v[0]` is not modified by the assignment to `u[0]` in `assign_first`.

In order to take this knowledge into account in our analyses, we analyze subprograms with pointers differently from subprograms without pointers. For every parametric region for a subprogram, we simulate an array parameter of the proper type for this region, indexed by pointers, which is returned after the subprogram completes as an additional result. Then, we adapt the function *equal* to these special arrays: like previously, *equal(x,x)* is true syntactically ; *equal(x,y)* is false if *x* and *y* belong to different regions ; *equal(x,y)* is *dont_know* if *x* and *y* belong to the same region.

A strict constraint to ensure soundness of the approach is that no two regions accessed in a subprogram should be equal for a particular call. This is the same condition as the one expressed by Reynolds (1978) in his seminal work on syntactic control of interference, where regions play in our case the role of collections in his work. This is guaranteed by failing to compute contextual regions if calling a subprogram leads to passing twice the same region in parameter.

```
1  void bad_regions(integer[] u, integer[] v) {
2    u[0] := v[0] + 1;
3  }
4
5  void main() {
6    integer[] loc = new integer(1); bad_regions(loc,loc);
7  }
```

Function `bad_regions` illustrates why this case should not be allowed. The contextual analysis computes two parametric regions `u` and `v` in function `bad_regions`. These regions being different, it is possible to check that `bad_regions` ensures postcondition *u[0] = v[0] + 1* by abstract interpretation or deductive verification, as described in Section 2. Then, call to `bad_regions` in `main` would violate this postcondition. This is not allowed, as this call passes the region for `loc` twice in parameter, as the current instance for the region of `u` and the region of `v`.

The contextual analysis thus trades incompleteness for precision and scaling: it either succeeds quickly with precise results, or it fails. Moreover, it is only guaranteed to be correct on complete programs, where all the calls are known.

4.2. Refining regions: a type-and-effect approach

The analysis by Hubert and Marché (2007) computes regions in a way that solves problem (i), context insensitivity, but still suffers from problem (ii), lack of modularity, while it adds a new problem (iii), incompleteness. We now propose a partial solution to both problems.

This analysis is mostly modular, as regions are computed in a modular way. Only the verification that different regions in a callee are not instantiated with the same region in the caller is not modular. We propose to delay this verification for incomplete programs by computing equivalent separation preconditions.

We start with computing for each region in a subprogram the set of pointers in this region that may be read or written, expressed in terms of variables at subprogram entry. This can be done with a flow-sensitive or a flow-insensitive analysis in a straightforward way. For example, the set of pointers read associated to region v in `bad_regions` is precisely v, and the set of pointers written associated to region u in `bad_regions` is precisely u. This must take into account memory accessed indirectly through a call. Then, we simply collect all effects of a function due to its memory accesses and calls.

Functions are processed in reverse topological order of the call-graph, and recursive calls are handled by iterating on strongly connected components until a fix-point is reached.

Once effects of a function are computed, the following precondition guarantees soundness of the contextual analysis: for each pair of locations possibly accessed in different regions, when one of these regions is possibly written, we generate a separation condition that guarantees they may not overlap, which amounts to having disequality preconditions between pointers. As already noted by Reynolds (1978), asking for the separation of all pairs of regions is overly restrictive. Regions that are only read cannot interfere, thus there is no need to ask for separation of these read-only regions, or *passive phrases* in Reynolds's terminology, which is what we do here. Furthermore, this is useful only for regions which may overlap, which means they should be of the same type. Notice that a region that is not accessible through subprogram parameters cannot be overlapping with any other region. This corresponds to a region allocated inside the subprogram, which is either an internal region, *i.e.*, it does not escape the subprogram, or a region reachable only through the subprogram result. Thus, no separation precondition is ever generated for pointers in such a region.

To see how this works, we recall the code of function `bad_regions`:

```
1  void bad_regions(integer[] u, integer[] v) {
2    u[0] := v[0] + 1;
3  }
```

On this function, we compute effects

$$writes(\rho_u) = \{u\}, reads(\rho_v) = \{v\}$$

which leads to the precondition

$$u \neq v.$$

We recall also the code of function `assign_first`:

```
1  void assign_first(integer[] u, integer[] v) {
2    u[0] := 0;
3  }
```

On this function, we compute effects

$$writes(\rho_u) = \{u\}$$

which leads to no precondition.

Separation preconditions make the contextual alias analysis completely modular, thus solving problem (ii), but it does not completely solve problem (iii), incompleteness of the contextual alias analysis. Indeed, the contextual analysis may still wrongly assume separation between regions. It is the case for subprogram `swap` below.

```
1  void swap(integer[] u, integer[] v) {
2    integer tmp; tmp := u[0]; u[0] := v[0]; v[0] := tmp;
3  }
```

For this subprogram, our analysis generates the precondition

$$u \neq v,$$

while it would still be correct to call `swap(x,x)`.

For those cases where our technique generates stronger separation conditions than necessary, a solution would be to manually annotate function parameters with explicit regions, much as what is done in Cyclone (Jim et al., 2002).

5. Experiments

The techniques described in this paper have been implemented in Frama-C, an open-source platform for the modular analysis of C programs. Target C programs are translated to an intermediate language on which we infer subprogram contracts, before verification conditions (VC) are generated

and sent to automatic provers to prove the safety of the original C programs. In the following, provers are summarized by their initial letter: A for Alt-Ergo v0.8, S for Simplify v1.5.4, Y for Yices v1.0.16 and Z for Z3 v1.3.

In Section 5.1, we describe the results of applying our tool to check the safety of an available string library. In Section 5.2, we describe the results of applying our tool to discriminate between unsafe and patched versions of open-source programs with vulnerabilities. The verification was performed completely automatically and modularly, with each function analyzed separately. In both cases, we manually added `FRAMA_C_STRING` user annotations in the code, which are predicates relating to our C memory model that indicate when a given parameter points to a string.

5.1. MINIX 3 standard string library

MINIX 3 is an open-source operating system designed to be highly reliable, flexible, and secure. To reach these goals, its code is intentionally small and simple. In particular, it implements a library for strings in an idiomatic and straightforward style, quite closely following the C standard. *E.g.*, here is the code for function `strcpy`:

```
1  char *strcpy(char *ret, const char *s2) {
2    char *s1 = ret;
3    while (*s1++ = *s2++)
4      /* EMPTY */ ;
5    return ret;
6  }
```

Overall, the standard C library for strings specifies 22 functions, all of which are implemented in the MINIX 3 library. We applied the different annotation inference methods described in this paper to check the safety of 21 of these functions (`strtok` is not treated because its safety relies on a proper sequencing of successive calls by the client program).

We compared 4 runs of the benchmark:

(1) *no annotation inference* - No annotation at all is inferred.
(2) *abstract interpretation* - Loop invariants are inferred by abstract interpretation, following algorithm ForwAbsInt.
(3) *quantifier elimination* - First, loop invariants are inferred by abstract interpretation, and then preconditions are inferred by quantifier elimination, following algorithm InferElim.
(4) *weakest preconditions* - First, loop invariants are inferred by abstract interpretation, and then preconditions are inferred by weakest preconditions and quantifier elimination. We tried both algorithms InferStrong and InferWeak, which gave the exact same results, due to the small size of source programs.

Results for these runs are summarized in Figs. 9–12. The total number of verification conditions (VC) varies between runs as it depends on the annotations inferred. As expected, automatic provers succeed in proving more verification conditions when more annotations are inferred. When the most precise method is used, prover Z3 manages to prove all verification conditions.

Not surprisingly, the total time elapsed decreases with the number of annotations inferred, as shown in Fig. 13. Indeed, in this case, the time spent for inferring annotations is far smaller than the time spent trying to prove unprovable verification conditions.

With the most precise inference method, a satisfying sufficient precondition to ensure the safety of each function is inferred. A precondition is satisfying when it is not too strong, so that usual patterns of usage for this function are allowed. *E.g.*, , the precondition for `strcpy` expresses that its arguments cannot alias and that the target buffer is large enough to hold the source string being copied, which translates here into:

$$(ret \neq s2) \wedge (0 \leq strlen(s2) < array\text{-}length(ret)).$$

The actual precondition we infer for `strcpy` is slightly more involved in reality, as our memory model for C takes into account the possibility of pointer arithmetic (Moy, 2009).

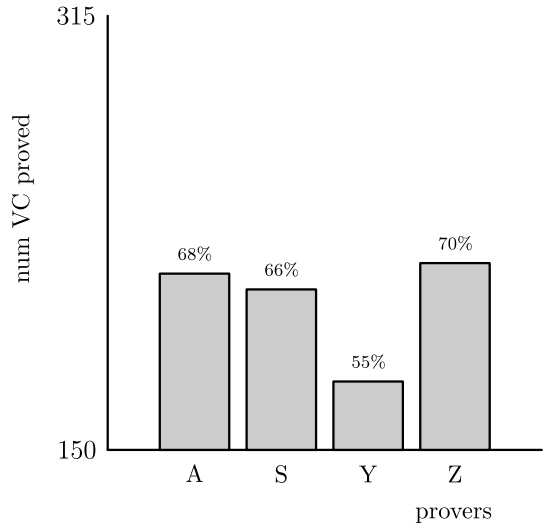


Fig. 9. No annotation inference.

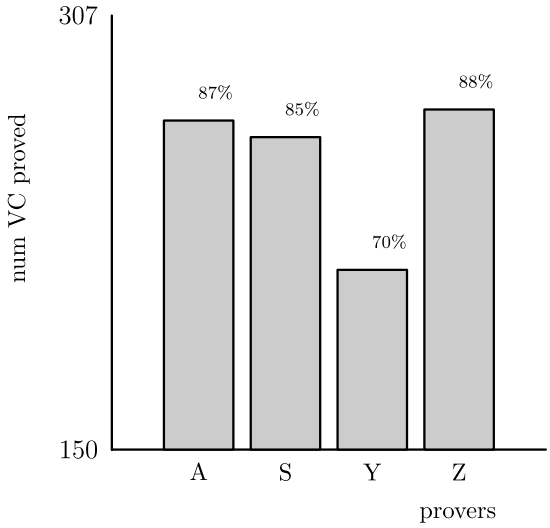


Fig. 10. Abstract interpretation.

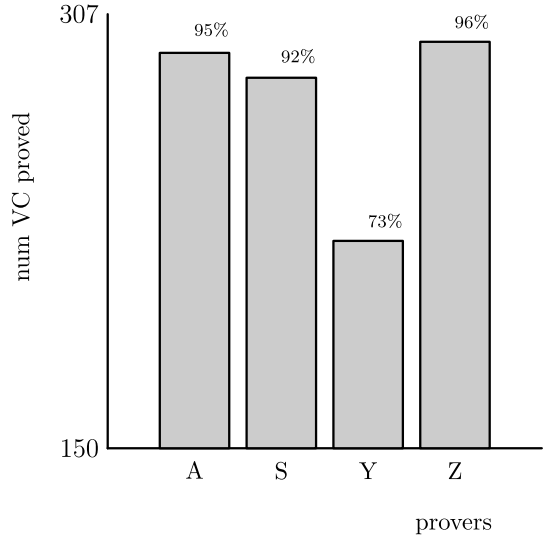


Fig. 11. Quantifier elimination.

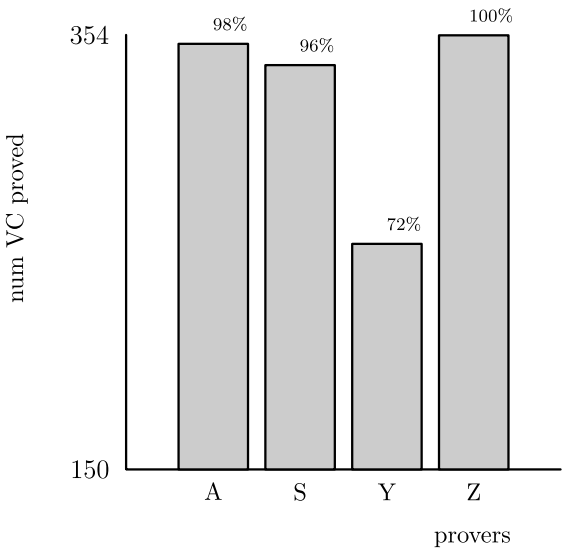


Fig. 12. Weakest preconditions.

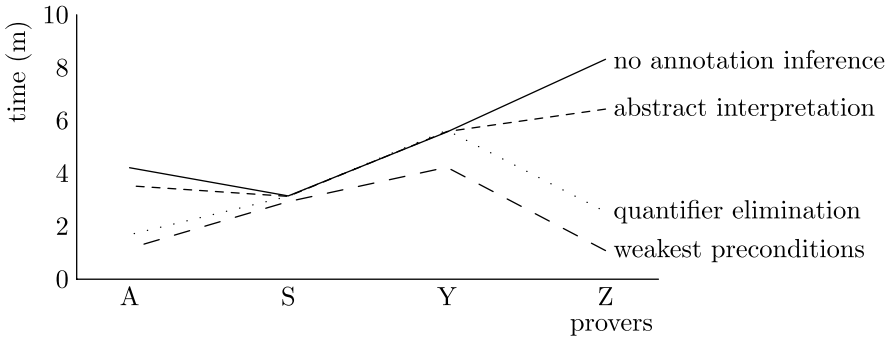


Fig. 13. Total time elapsed.

5.2. Benchmarks of vulnerabilities

We ran our tool on two benchmarks of real code vulnerabilities, the Verisec Suite and Zitser's benchmark. These benchmarks consist in snippets of open-source code containing buffer overflow vulnerabilities, together with their patched versions. Vulnerabilities are identified by their CVE number (ex: CVE-2004-0940). They are extracted from popular open-source server programs such as Apache, Samba and sendmail. Variations over each vulnerability are presented as a set of "bad" and "ok" snippets of code, usually in pairs, so that each "ok" version corresponds to the patch of a "bad" version. The two benchmarks differ in the number and difficulty of snippets:

- The Verisec Suite (Ku et al., 2007) targets 22 vulnerabilities in 12 programs, for a total of 144 "bad" and 140 "ok" snippets of code. Each snippet has a size between 16 and 233 loc, with an average of 69 loc, not counting include files.
- Zitser's benchmark (Zitser et al., 2004) targets 14 vulnerabilities in 3 programs, with a "bad" and an "ok" snippet of code for each. Each snippet has a size between 218 and 777 loc, with an average of 506 loc, not counting include files.

As expected from these figures, Zitser's benchmark is more difficult to verify than the Verisec Suite, while the latter allows a finer analysis of results due to its large number of snippets with small variations. Although these snippets come equipped with a main function, we do not perform any global analysis on programs. Indeed, our target is to test the performance of our techniques to check safety both automatically and modularly, not needing the complete program. Thus, each function is analyzed independently of its calling context, in reverse topological order of the call-graph (*i.e.*, leaf functions first).

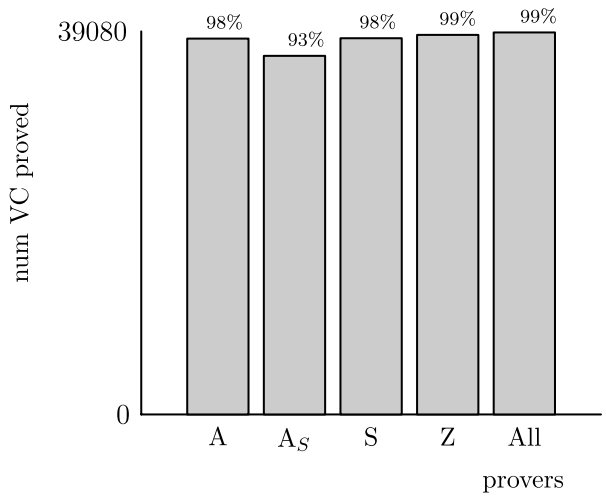


Fig. 14. Proof results (105/140).

First, we added `FRAMA_C_STRING` user annotations in the code, indicate when a given parameter points to a string. Overall, we added 389 such annotations denoting those parameters, returns and variables that should be strings. For some cases where our inference technique does not generate precise enough annotations, we selectively added manual annotations in the code, in the form of assertions.

We did not complete this process until all VC are proved, for lack of time. During this process, we found 16 bugs in the "ok" snippets, where a buffer was either incorrectly not null-terminated, or a buffer possibly accessed beyond its bounds. We reported the corresponding patches to the authors of the suite, which recognized their suite was mostly designed to distinguish between "ok" and "bad" accesses at one particular point in each program, not necessarily granting safety of "ok" snippets.

For these experiments, we selected the following set of options:

- *provers Alt-Ergo v0.8, Simplify v1.5.4 and Z3 v1.3*: these are the provers that perform best on our verification conditions, denoted A, S and Z respectively. We also run Alt-Ergo under another setting, denoted A_S, which applies some heuristics for filtering hypotheses, focusing proof search on the goal and consequently allows one to find easy proofs more quickly (Couchot and Hubert, 2007; Conchon and Contejean, 2008).
- *annotation inference* InferStrong : it is the most precise annotation inference method that scales to these functions of up to 200 loc with many conditions and loops. The cheaper ForwAbsInt and InferElim are not precise enough in many cases, and the more precise InferWeak does not scale.
- *abstract domain of octagons*: it is the most appropriate abstract domain for these tests, whose safety essentially depends on relations between pairs of variables. It is cheaper than the abstract domain of polyhedrons, and it leads to better widening results in many cases.

Due to current limitations in our tool, only 105 "ok" snippets out of 140 can be analyzed. On these snippets analyzed, Fig. 14 shows that, while 99.7% of VC are proved automatically (38,968 VC for a total of 39,080), 112 VC are not proved automatically. Overall, the snippets fall into the following cases:

- 35 snippets cannot be analyzed: 14 tests reach the allowed memory bound (0.25 M) or time bound (10 mn) during generation of annotations, 1 contains backward goto, the remaining ones trigger limitations in our implementation;
- 78 snippets are completely proved;
- 27 snippets are partly proved. Fig. 17 shows that, among these tests, a majority have only a few unproved VC, which could be dealt with manually, either to add intermediate assertions to help provers, or by review.

Fig. 15 presents the total running time for all four provers, which only exceeds 3 h for Alt-Ergo with hypotheses filtering, because some goals then become unprovable, which causes the prover to reach the 10 s timeout while searching for an impossible proof.

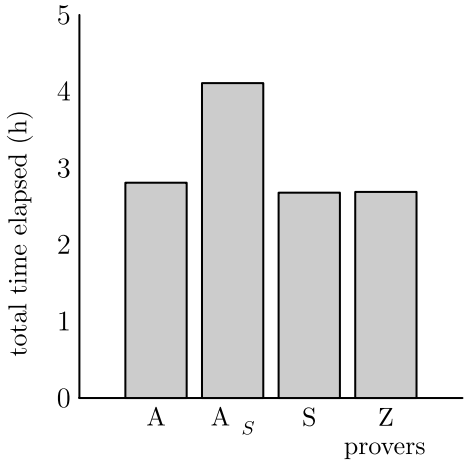


Fig. 15. Time results (105/140).

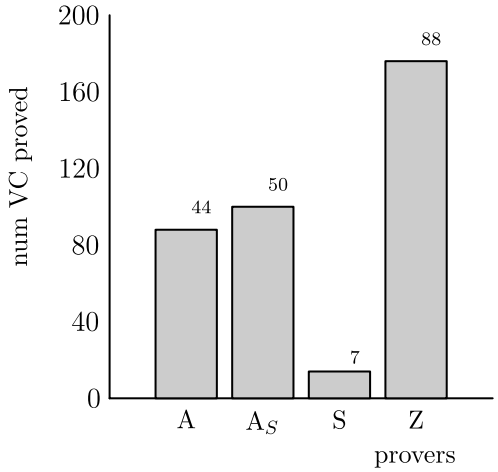


Fig. 16. Provers' strength (105/140).

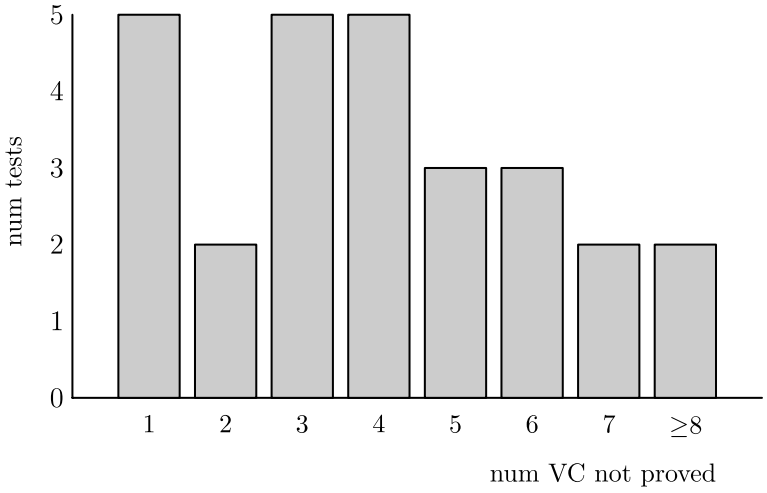


Fig. 17. Unproved VC (27/140).

Fig. 16 shows why it is in general a good idea to use a combination of provers rather than a single prover. It presents the number of VC that each prover is the only one to prove. Notice that using all 4 provers, including Alt-Ergo with hypotheses filtering, is indeed mandatory to decrease the number of unproved VC.

Figs. 18 and 19 present the number of relations ((dis-)equalities and inequalities) in annotations inferred. Columns Pre and Post report the number of relations in preconditions and postconditions

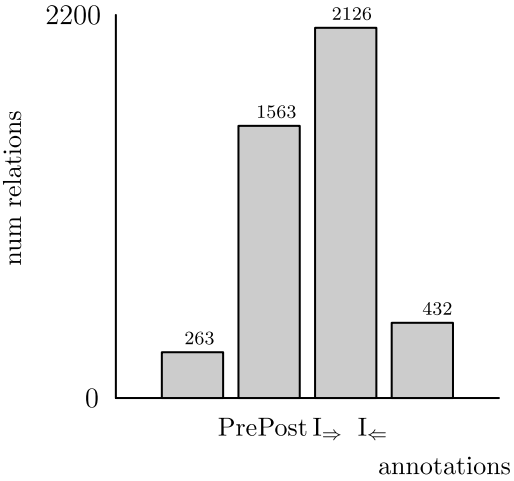


Fig. 18. Verisec annotations.

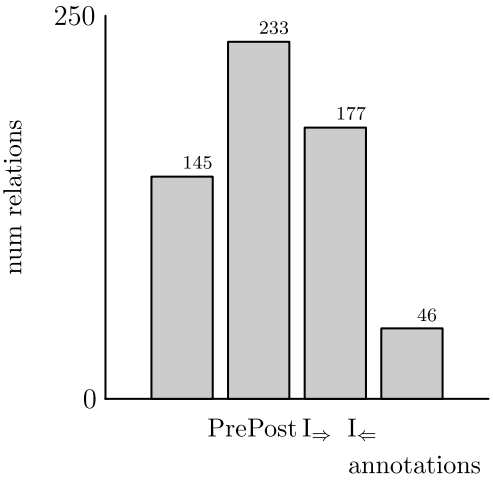


Fig. 19. Zitser annotations.

inferred, while columns $I_{\Rightarrow}$ and $I_{\Leftarrow}$ report the number of relations in loop invariants inferred respectively by forward abstract interpretation and the backward InferStrong algorithm. Formulas in $I_{\Leftarrow}$ do not repeat those found in $I_{\Rightarrow}$. The number of relations is larger in postconditions because they are built as a disjunction of cases for each return statement in the source program, and in loop invariants because they mention local variables, both from the source program and generated by our instrumentation.

6. Related work

6.1. Loop invariant inference

Historically, array bound checking has been one of the first difficult properties about programs that people tried to prove, the hardest part of the verification task being the automatic inference of loop invariants. Cousot and Halbwachs (1978) applied abstract interpretation over polyhedrons and managed to check memory safety of an implementation of *heap sort*, using manual preconditions. Suzuki and Ishihata (1977) devised a method based on weakest preconditions to check memory safety of an implementation of *tree sort*. They used Fourier–Motzkin elimination at loop entrance as a heuristic to make their induction-iteration method converge. There are examples (Moy, 2009) where abstract interpretation outperforms induction-iteration or the opposite. By combining abstract interpretation and weakest preconditions, we manage to outperform both in many cases.

More recently, Xu et al. (2000) and Xu (2000) have refined with success the induction-iteration method for safety checking of machine code. They use forward abstract interpretation and induction-iteration separately to generate loop invariants, and they rely on user preconditions to provide a valid calling context.

More generally, a large number of works have targeted loop invariant inference by abstract interpretation, predicate abstraction and weakest preconditions/strongest postconditions, or a combination thereof. This should continue to be a major research goal in the years to come. Promising techniques combine abstract interpretation and deductive verification. Leino and Logozzo (2005) build a real feedback loop between a theorem prover and an abstract interpretation module to generate loop invariants. Leino and Logozzo (2007) present an embedding of the abstract interpretation technique of widening inside a theorem prover. The opposite approach of performing abstract interpretation on logic formulas has been presented by Gulwani and Tiwari (2006).

6.2. Precondition inference

Bourdoncle (1993) defines abstract debugging as backward abstract interpretation from assertions. Along the way, he generates loop invariants and preconditions in order to prove these assertions. He focuses on array bound checking too. His backward propagation merges the conditions to reach the program point where the assertion is checked and the conditions to make this assertion valid. The dual approach of propagating backward a superset of the forbidden states is described by Rival (2005). We have shown in this chapter the limitations of these approaches.

Gulwani and Tiwari (2007) consider the problem of assertion checking for the special case of equalities in a restricted language with only non-deterministic branching. Using a method based on unification, they manage to generate necessary and sufficient preconditions for assertions to hold. Unfortunately, unification does not work for the relations that arise most often in practice for safety checking, namely less-than and greater-than relations. Our method only generates sufficient preconditions, but it applies to those arithmetic relations found in practice.

In a recent article, Popeea et al. (2008) describe a technique similar to ours to generate sufficient preconditions. They combine forward abstract interpretation with constraint solving to generate preconditions for optimization of C programs.

The tool Houdini (Flanagan and Leino, 2001) considers a large number of candidate invariants at various program points, based on the syntactic structure of the program, and it filters out spurious invariants by checking if they hold in the program considered with ESC/Java. A problem with this approach is that it may infer preconditions that are too strong, based on the calling contexts observed in the program rather than based on the function being analyzed. A big limitation compared to our approach is that they cannot infer disjunctive preconditions.

6.3. Alias analyses based on regions

The possibility of dividing memory into regions with the results of a type-based alias analyses dates back to the work of Talpin and Jouvelot on higher-order functional languages (Talpin and Jouvelot, 1991, 1992). The purpose of their work is to compute effects, so overlapping between regions is allowed, which makes it easy to compute regions with a context-sensitive analysis. Tofte and Talpin apply this analysis to perform static memory allocation (Tofte and Talpin, 1997). Again, overlapping between regions is not a concern.

Steensgaard's alias analysis (Steensgaard, 1996) is the global context-insensitive counterpart of Talpin's local context-sensitive analysis. It is a real alias analysis, meaning that different regions truly cannot overlap. It is the best known scalable alias analysis, but, being dereferencing-insensitive, its precision is low. Liang and Harrold present a context-sensitive variant of Steensgaard's alias analysis to improve its precision (Liang and Harrold, 2001; Lattner et al., 2007). Contrary to Talpin's analysis, they merge callee's regions when they correspond to the same region in the caller, possibly losing some precision. Hubert and Marché have chosen instead to fail in this case, thus gaining in precision at the cost of completeness (Hubert and Marché, 2007). We manage to retain this precision and still be complete by generating function contracts to be checked by deductive verification.

6.4. Alias control techniques

Alias control techniques have been pioneered by Reynolds in his work on syntactic control of interference (Reynolds, 1978), where collections play the role of regions in our work. This notion has

been granted a keyword, `restrict` in C99 standard (C99, 2000), that conveys the programmer's "guarantee" that a pointer is the unique reference on some memory. Various authors have described annotation-based systems to help programmers specify pointer separation (Assaad and Leavens, 2001; Aiken et al., 2003; Koes et al., 2004).

6.5. Heap and shape analyses

It may come as a surprise that we do not need a deeper understanding of the heap to analyze programs with lists, trees, or other pointer-based data structures. This is because we only consider, in our experiments, safety checking, which is not so much concerned with the shape of the heap, contrary to program termination and verification of behavioral properties. In particular, we do not relate to separation logic or shape analysis. Calcagno et al. (2007) present an analysis to infer sufficient preconditions for list manipulating programs.

6.6. Verification of string libraries

In his Master's thesis (Starostin, 2006), Starostin fully verified a string library he implemented in C0. While our work focuses on automatic verification of safety only, his work is a manual verification inside Isabelle/HOL of the complete behavior of functions. Also, he codesigned the implementation and the proof, while we want to check the safety of existing libraries.

In his PhD thesis (Norrish, 1998), Norrish presents a complete verification of the behavior of function `strcpy` as implemented by Kernighan and Ritchie (1978). It is in fact the same as the one still implemented in most systems, like the one in MINIX 3 presented in Section 5.1. His work is a manual verification inside HOL based on a deep embedding of C semantics, but he still manages to automate the proof of some properties involving arithmetic, most notably safety properties and separation properties. However, he notices the poor performance of automated techniques in HOL on the particular verification goals he generates.

6.7. Verification of benchmarks of vulnerabilities

Zitser's benchmark has had a great influence on the design of tools for safety checking of C programs. This was partly due to the integration of Zitser's programs in the SAMATE Reference Dataset used to compare tools for software assurance. By showing in their study (Zitser et al., 2004) that none of the five modern static analysis tools tested was better than a random choice when discriminating between an unsafe program and its patched version, Zitser et al. have set a milestone for such tools. Since then, various tools have claimed to be able to improve on their results:

- Hackett et al. (2006) present a tool based on SAL lightweight annotations that succeeds in discriminating most of Zitser's test cases. However, since they use unsound static analysis techniques, they cannot make any claim about the safety of the patched programs.
- Chaki and Hissam (2006) present a tool based on software model checking that improves on the confusion rate. They obtain that whenever their tool detects a potential buffer overflow in an unsafe program, it proves safety of the same buffer access in the patched version. Unfortunately, their tool also has lower detection and resolution rates than the two best tools presented in the study of Zitser et al., namely PolySpace and Splint.

The Verisec Suite was developed with the same interface as Zitser's benchmark, to provide many simpler and diverse examples more amenable to verification. In particular, it makes it easier to bound the size of inputs for model checkers. Hart et al. manage to discriminate 49 tests out of 59 taken from the Verisec Suite by applying template-based model checking, where models of the program invariants are given by the programmer (Hart et al., 2008). Like Hackett et al., they only report their results on these identified potential overflows in the unsafe programs, not on all possible overflows. Contrary to our work, they perform a global analysis that takes profit from the simple crafted calling context of functions. In particular, they exploit the small bound on the size of buffers, which is expected in software model checking.

7. Conclusion and perspectives

Verifying, formally, specified behavioral properties of programs is currently too costly to be widely applied in software industry. Our work proposes solutions to the main issue which prevents this approach from scaling up, the insertion of annotations in the code. The techniques we propose to automatically generate annotations discharge the user from manually annotating all subprograms. We think in particular that our generated preconditions are more precise than everything done before. The modularity of the method allows one to apply it on small pieces of code independently, and in particular on reusable software libraries.

We think that building *verified libraries* is a key to allowing formal behavioral specifications to spread up in industry. To reach this goal, there are other issues to be solved.

- Our experimentations mainly focused on safety properties and not on user-defined behavioral properties. In our approach we deal with subprograms by traversing the call graph from the bottom (the "leaf" subprograms) to the top. However, if one wants the establish a given behavioral property of the main program, it is desirable to design a method for propagation of given user annotations from the main program entry to subprograms. Such an idea has been investigated by Rousset (2008) with *interprocedural* generation of annotations.
- Heap analysis is a very active area of research in static safety checking. It consists in inferring and checking invariants on the heap structure. Separation logic is a the key technology enabling these successes. As more decision procedures are built for separation logic, it might be profitable to generate the separation preconditions generated by our modular inference of region in this logic rather than as conjunctions of differences between pointers, as we do currently.

Acknowledgements

We would like to thank K. Rustan, M. Leino and Xavier Leroy for their insightful comments on Moy's PhD thesis which was the basis for this paper.

References

Aiken, A., Foster, J. S., Kodumal, J., Terauchi, T., 2003. Checking and inferring local non-aliasing. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 129–140.

Assaad, M.G., Leavens, G.T., 2001. Alias-free parameters in C for better reasoning and optimization. Technical Report 01-11, Department of Computer Science, Iowa State University.

Barnett, M., Leino, K.R.M., Schulte, W., 2004. The Spec# Programming System: An Overview. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. CASSIS'04. In: Lecture Notes in Computer Science, vol. 3362. Springer, pp. 49–69.

Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V., 2008. ACSL: ANSI/ISO C Specification Language. http://frama-c.cea.fr/acsl.html.

Bourdoncle, F., 1993. Assertion-based debugging of imperative programs by abstract interpretation. In: ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering. Springer-Verlag, London, UK, pp. 501–516.

Bradley, A.R., Manna, Z., 2007. The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E., 2004. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer.

C99,, 2000. ISO/IEC 9899:1999: Programming Languages – C. International Organization for Standardization.

Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H., 2007. Footprint analysis: a shape analysis that discovers preconditions. In: Proceedings of the 14th International Static Analysis Symposium. In: Lecture Notes in Computer Science, vol. 4634. Springer-Verlag, pp. 402–418.

Chaki, S., Hissam, S., 2006. Certifying the absence of buffer overflows. Technical Note CMU/SEI-2006-TN-030, Carnegie-Mellon University/Software Engineering Institute.

Colon, M., Sankaranarayanan, S., Sipma, H., 2003. Linear invariant generation using non-linear constraint solving. In: Proc. of the Int. Conf. on Computer Aided Verification, CAV. In: Lecture Notes in Computer Science, vol. 2725. pp. 420–432.

Conchon, S., Contejean, E., 2008. The Alt-Ergo automatic theorem prover http://alt-ergo.lri.fr/.

Couchot, J.-F., Hubert, T., 2007. A Graph-based Strategy for the Selection of Hypotheses. In: FTP 2007 — International Workshop on First-Order Theorem Proving. Liverpool, UK.

Cousot, P., Cousot, R., 1979. Systematic design of program analysis frameworks. In: POPL'79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 269–282.

Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program. In: POPL'78: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 84–96.
Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W., 2009. VCC: Contract-based modular verification of concurrent C. In: 31st International Conference on Software Engineering. ICSE 2009, May 16–24, Vancouver, Canada. Companion Volume. IEEE, pp. 429–430.
Dijkstra, E.W., 1976. A Discipline of Programming. In: Series in Automatic Computation, Prentice Hall Int.
Dill, D.L., 1989. Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits. MIT Press, Cambridge, MA, USA.
Filliâtre, J.-C., 2003. Verification of non-functional programs using interpretations in type theory. Journal of Functional Programming 13 (4), 709–745.
Filliâtre, J.-C., Marché, C., 2007. The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (Eds.), 19th International Conference on Computer Aided Verification. In: Lecture Notes in Computer Science, vol. 4590. Springer, Berlin, Germany, pp. 173–177. URL http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf.
Flanagan, C., Leino, K.R.M., 2001. Houdini, an annotation assistant for ESC/Java. In: FME'01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity. Springer-Verlag, London, UK, pp. 500–517.
Gulwani, S., Tiwari, A., 2006. Combining abstract interpreters. In: Annual ACM Conference on Programming Language Design and Implementation. ACM.
Gulwani, S., Tiwari, A., 2007. Assertion checking unified. In: The 8th International Conference on Verification, Model Checking and Abstract Interpretation. Springer, pp. 363–377.
Hackett, B., Das, M., Wang, D., Yang, Z., 2006. Modular checking for buffer overflows in the large. In: ICSE'06: Proceedings of the 28th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 232–241.
Hart, T.E., Ku, K., Lie, D., Chechik, M., Gurfinkel, A., 2008. Ptyasm: Software model checking with proof templates. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE'08.
Hatcliff, J., Leavens, G.T., Leino, K.R.M., Miller, P., Parkinson, M., 2009. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, a survey paper, Draft.
Hubert, T., Marché, C., Mar. 2007. Separation analysis for deductive verification. In: Heap Analysis and Verification, HAV'07. Braga, Portugal, pp. 81–93, http://www.lri.fr/~marche/hubert07hav.pdf.
Janota, M., 2007. Assertion-based loop invariant generation. In: Proceedings of the 1st International Workshop on Invariant Generation, WING'07. Hagenberg, Austria, workshop at CALCULEMUS, 2007.
Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y., 2002. Cyclone: A safe dialect of C. In: Proc. 2002 USENIX Annual Technical Conference. Berkeley, CA, USA, pp. 275–288.
Karr, M., 1976. Affine relationships among variables of a program. Acta Informatica 133–151.
Kernighan, B.W., Ritchie, D.M., 1978. The C Programming Language. Prentice-Hall, Englewood Cliffs, New Jersey.
Koes, D., Budiu, M., Venkataramani, G., 2004. Programmer specified pointer independence. In: MSP'04: Proceedings of the 2004 Workshop on Memory System Performance. ACM, New York, NY, USA, pp. 51–59.
Ku, K., Hart, T.E., Chechik, M., Lie, D., 2007. A buffer overflow benchmark for software model checkers. In: ASE'07: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 389–392.
Lattner, C., Lenharth, A., Adve, V., 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. SIGPLAN Notices 42 (6), 278–289.
Leino, K.R.M., Logozzo, F., 2005. Loop invariants on demand. In: APLAS'05: Proceedings of The 3rd ASIAN Symposium on Programming Languages and Systems. In: LNCS, Springer-Verlag, pp. 119–134.
Leino, K.R.M., Logozzo, F., 2007. Using widenings to infer loop invariants inside an SMT solver, or: a theorem prover as abstract domain. Tech. Rep. RISC-Linz Report Series No. 07-07, RISC, Hagenberg, Austria, proc. WING'07.
Leino, K.R.M., Saxe, J.B., Stata, R., 1999. Checking Java programs via guarded commands. In: Proceedings of the Workshop on Object-Oriented Technology. Springer-Verlag, London, UK, pp. 110–111.
Liang, D., Harrold, M.J., 2001. Efficient computation of parameterized pointer information for interprocedural analyses. In: SAS'01: Proceedings of the 8th International Symposium on Static Analysis. Springer-Verlag, London, UK, pp. 279–298.
Marché, C., 2007. Jessie: an intermediate language for Java and C verification. In: Programming Languages meets Program Verification (PLPV). ACM, Freiburg, Germany, pp. 1–2. URL http://doi.acm.org/10.1145/1292597.1292602.
Miné, A., 2006. The octagon abstract domain. Higher Order Symbolic Computation 19 (1), 31–100.
Monniaux, D., 2008. A quantifier elimination algorithm for linear real arithmetic. CoRR abs/0803.1575, informal publication.
Moy, Y., 2009. Automatic modular static safety checking for C programs. Ph.D. Thesis, Université Paris-Sud.
Norrish, M., 1998. C formalised in HOL. Ph.D. Thesis, University of Cambridge.
Popeea, C., Xu, D.N., Chin, W.-N., 2008. A practical and precise inference and specializer for array bound checks elimination. In: PEPM'08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. ACM, New York, NY, USA, pp. 177–187.
Reynolds, J.C., 1978. Syntactic control of interference. In: POPL'78: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 39–46.
Rival, X., 2005. Understanding the origin of alarms in ASTRÉE. In: 12th Static Analysis Symposium. SAS'05. In: LNCS, vol. 3672. Springer-Verlag, London, UK, pp. 303–319.
Rousset, N., 2008. Automatisation de la spécification et de la vérification d'applications Java Card. Thèse de doctorat, Université Paris-Sud.
Starostin, A., 2006. Formal verification of a C-library for strings. Master's Thesis, Saarland University.
Steensgaard, B., 1996. Points-to analysis in almost linear time. In: POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 32–41.
Suzuki, N., Ishihata, K., 1977. Implementation of an array bound checker. In: POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 132–143.
Talpin, J.-P., Jouvelot, P., 1991. Polymorphic type region and effect inference. Tech. Rep. EMP-CRI E/150.

Talpin, J.-P., Jouvelot, P., 1992. The type and effect discipline. In: Seventh Annual IEEE Symposium on Logic in Computer Science. Santa Cruz, California. IEEE Computer Society Press, Los Alamitos, California, pp. 162–173.
Tofte, M., Talpin, J.-P., 1997. Region-based memory management. Information and Computation.
Weispfenning, V., 1997. Complexity and uniformity of elimination in presburger arithmetic. In: ISSAC'97: Proceedings of the 1997 International Symposium on Symbolic and Algebraic computation. ACM, New York, NY, USA, pp. 48–53.
Xu, Z., 2000. Safety checking of machine code. Ph.D. Thesis, Univ. of Wisconsin, Madison.
Xu, Z., Miller, B.P., Reps, T., 2000. Safety checking of machine code. ACM SIGPLAN Notices 35 (5), 70–82.
Zitser, M., Lippmann, R., Leek, T., 2004. Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT Software Engineering Notes 29 (6), 97–106.