INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Checking C Pointer Programs for Memory Safety

Yannick Moy

N° ????

July 2007

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

Checking C Pointer Programs for Memory Safety

Yannick Moy*†‡† ‡

Thème SYM — Systèmes symboliques
Projets ProVal

Rapport de recherche n ???? — July 2007 — 51 pages

Abstract: We propose an original approach for checking memory safety of C pointer programs possibly including pointer arithmetic and sharing (but no casts, structures, double indirection or memory deallocation). This involves first identifying aliasing and strings, which we do in a local setting rather than through a global analysis as it is done usually. Our separation analysis in particular is a totally new treatment of non-aliasing. We present for the first time two abstract lattices to deal with local pointer aliasing and local pointer non-aliasing in an abstract interpretation framework. The key feature of our work is to combine abstract interpretation techniques and deductive verification. The approach is modular and contextual, thanks to the use of Hoare-style annotations (pre- and postconditions), allowing to verify each C function independently. Abstract interpretation techniques are used to automatically generate such annotations, in an idiomatic way: standard practice of C programming is identified and incorporated as heuristics. Abstract interpretation and deductive verification are both used to check these annotations in a sound way. Our first contribution is the design of an abstract domain for implications, which makes it possible to build efficient contextual analyses. Our second contribution is an efficient back-and-forth propagation method to generate contextual annotations in a modular way, in the framework of abstract interpretation. Thanks to previously unknown loop refinement operators, this propagation method does not require iterating around loops. We implemented our method in Caduceus, a tool for the verification of C programs, and successfully verified automatically the C standard string library functions.

Key-words: C programming language, abstract interpretation, deductive verification, pointer programs, aliasing, memory safety

* France Télécom, Lannion, F-22307
† INRIA Futurs, ProVal, Parc Orsay Université, F-91893
‡ Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 — Télécopie : +33 1 60 19 66 08

Sûreté mémoire des programmes C à pointeurs

Résumé : Nous présentons une approche originale pour vérifier la sûreté des accès mémoires des programmes C à pointeurs pouvant contenir de l'arithmétique de pointeurs et du partage (mais pas de casts, de structures, de double indirection ou de libération mémoire). Cela consiste en premier lieu à identifier le partage mémoire et les chaînes de caractères, ce que nous faisons dans un contexte local plutôt que par une analyse globale comme c'est le cas habituellement. Notre analyse de séparation mémoire en particulier est complètement nouvelle. Nous présentons pour la première fois deux treillis pour l'interprétation abstraite qui permettent de traiter le partage et le non-partage mémoire de façon locale. La principale caractéristique de notre analyse est la combinaison de l'interprétation abstraite et de la vérification par preuve. Notre approche est modulaire et contextuelle, grâce à l'utilisation d'annotations à la Hoare (pré- et post-conditions), ce qui permet d'analyser chaque fonction de manière séparée. On utilise les techniques d'interprétation abstraite pour générer automatiquement ces annotations, de façon idiomatique: les pratiques standard de programmation en C sont identifiées et incorporées comme des heuristiques. L'interprétation abstraite et la vérification déductive sont toutes deux utilisées pour vérifier ces annotations de manière sûre. Notre première contribution est la création d'un domaine abstrait pour les implications, ce qui rend possible la construction d'analyses contextuelles efficaces. Notre seconde contribution est une méthode de propagation arrière-avant efficace qui génère des annotations contextuelles de façon modulaire, dans le cadre de l'interprétation abstraite. Grâce à des opérateurs de raffinement originaux, cette méthode de propagation ne nécessite pas d'itérer autour des boucles du programme. Nous avons implémenté cette méthode dans Caduceus, un outil pour la vérification de programmes C, ce qui nous a permis de vérifier automatiquement les fonctions de la librairie C standard de manipulation des chaînes de caractères.

Mots-clés : Langage C, interprétation abstraite, méthodes déductives, vérification, programmes à pointeurs, partage mémoire, sûreté mémoire

Contents

1 Introduction **4**
1.1 Rationale . . . 4
1.2 Useful techniques that prove . . . 4
1.2.1 Abstract interpretation . . . 4
1.2.2 Deductive verification . . . 6
1.2.3 Common language . . . 6
1.3 Context of interest . . . 7
1.4 Target language . . . 8
1.5 Method overview and motivating example . . . 9

2 Aliasing and strings **10**
2.1 Local aliasing . . . 11
2.2 Local non-aliasing . . . 14
2.3 Pointers and strings . . . 18

3 Modular and contextual verification **19**
3.1 Verification frameworks . . . 19
3.2 Memory model . . . 20
3.3 Meta-variables and uninterpreted functions . . . 24
3.4 Implication lattice . . . 25
3.5 Inferring loop invariants and function preconditions . . . 30
3.5.1 Initial forward propagation . . . 30
3.5.2 Starting point: memory safety conditions . . . 31
3.5.3 Backward one-pass propagation . . . 31
3.5.4 Loop elimination . . . 32
3.5.5 Contextual one-pass forward propagation . . . 34
3.5.6 Flashback: the case of strings . . . 36
3.6 Proving memory safety . . . 37
3.7 Adding useful predicates to context . . . 38
3.8 Overall method . . . 38

4 Implementation **39**

5 Experiments **42**

6 Related work **43**

7 Future work **46**

8 Conclusion **47**

1 Introduction

1.1 Rationale

This work originated from the observation that no currently available tool can guarantee the absence of threat on memory safety on real C programs used in embedded devices. Although experienced C programmers can easily review small to medium programs for memory safety, those tools are defeated by quite simple programs, with few aliasing and no casts. The main reasons for this situation are the lack of support in these tools for widely spread C idioms, as well as the programmers' ability to understand comments embedded in the code, that reveal some invariants or preconditions for the programs to be correct.

Our primary goal is to address the first of these concerns, to increase the tools knowledge of idiomatic C, in a setting which already allows progress on the second concern by giving programmers the ability to formally specify their additional knowledge of the programs' logic. The subset of C that we target includes those constructs that usually make the verification task difficult: pointer arithmetic, dynamic allocation, aliasing and strings.

Our secondary goal is to allow real modular verification of programs, that fits as much as possible the programmers' understanding of modularity. While most tools base their modular verification on some *best guess* for stubbing the missing parts of programs, a more human-oriented modularity would focus on the assumptions needed to make programs correct. This is what we do.

Our third and last goal supports the realization of modular verification, where the basic modules of a C program are its functions. Many C functions can be called in a number of valid contexts (think of the nullity of pointer arguments), with different behaviours and different assumptions. Merging these contexts makes no sense, which is why context-sensitivity is a key feature for verifying these programs. Classically, the context of interest is taken to be the calling context, in a top-down analysis of the call-graph. Our approach focuses on the contexts needed by a function's body to make programs correct, in a bottom-up approach.

As a general rule, we do not emphasize generality but fit-for-purpose: our analysis should be able to verify the memory safety of large C programs used in real embedded software. Our method does not yet support casts, structures, double indirection and memory deallocation. Including any of these features in our target language will require new techniques in addition to the ones we present here.

1.2 Useful techniques that prove ...

... but also techniques that prove useful. We use a novel combination of powerful techniques: abstract interpretation and deductive verification.

1.2.1 Abstract interpretation

Abstract interpretation, or **AI** for short, is a technique to automatically build models of programs in some chosen directions of abstraction, which allows to infer many useful invari-

ants on the models that are true on the programs too. AI works by successive iterations on the program's control flow that collect the sets of values of variables in the model at each point.

An *abstract lattice* is an algebraic structure that describes the ordering relation in the model, so that each iteration can only increase a value in this ordering. A lattice is usually best described by a tuple $(L, \sqsubset_L, \bot_L, \top_L, \sqcup_L, \sqcap_L)$, where:

L is the set of elements in the lattice

$\sqsubset_L$ is the ordering relation

$\bot_L$ is the least element in the lattice

$\top_L$ is the greatest element in the lattice

$\sqcup_L$ is the union (join) of elements in the lattice

$\sqcap_L$ is the intersection (meet) of elements in the lattice

An important property is that $\sqcup_L$ and $\sqcap_L$ are consistent with respect to the ordering $\sqsubset_L$:

$$x \sqsubset_L y \Rightarrow (x \sqcup_L y = y \wedge x \sqcap_L y = x) \tag{1}$$

An *abstract domain* is an abstract lattice together with transfer functions that over-approximate the effect of a program statement on a value in this lattice. Of particular importance are the transfer functions for assignment and test. Abstract domains can be combined by a variety of techniques to get more precise information on the program than through separated analyses.

Connection between the program and its model is done through a Galois connection (α_L, γ_L), where the abstraction function α_L maps each set of concrete program states into an abstract element of L, and the concretization function γ_L 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 behaviours, we need the following properties on the concrete side:

$$\gamma_L(x \sqcup_L y) \supset \gamma_L(x) \cup \gamma_L(y) \tag{2}$$

$$\gamma_L(x \sqcap_L y) \subset \gamma_L(x) \cap \gamma_L(y) \tag{3}$$

$$\gamma_L \circ \alpha_L(s) \supset s \tag{4}$$

Most lattices we use in practice are stable by intersection, or convex. Then we can rewrite equation (3) to:

$$\gamma_L(x \sqcap_L y) = \gamma_L(x) \cap \gamma_L(y) \tag{5}$$

On domains with lattices of bounded height, convergence is ensured by the monotonicity of transfer functions. To accelerate convergence, or to ensure it in the general case, a widening operator can be provided. It is used to provide an over-approximation of the set of reachable states of the program.

1.2.2 Deductive verification

Deductive verification, or **DV** for short, is a technique to automatically or interactively prove or disprove validity of formulas in some chosen logic. We will be interested in formulas of first-order logic with quantifiers and integer arithmetic, which arise naturally when considering memory safety in C.

Assertions of interest about programs can be transformed into such formulas by computing Hoare-style weakest preconditions for these assertions. These are based on Hoare logic, that gives rules to reason about Hoare triples. A Hoare triple is a logical formula denoted $\{P\}\ i\ \{Q\}$, where P and Q are logical formulas in the underlying logic and i is a statement of the language found in the program. In the general case, we can use as atom in the underlying logic any side-effect free expression in our language. In our case, we will restrict atoms to side-effect free expressions based only on variables and a few arithmetic operators. Equation 6 defines validity for Hoare triples, where S_1 and S_2 are sets of states, $\xrightarrow{i}$ is the transition relation between sets of states for statement i and we write $S \models P$ when formula P is valid in the set of states S.

$$\{P\}\ i\ \{Q\} \equiv \forall S_1, S_2.\ (S_1 \xrightarrow{i} S_2) \wedge (S_1 \models P) \rightarrow (S_2 \models Q) \tag{6}$$

The following are valid Hoare triples:

$$\{x = 1\}\ x = x + 2;\ \{x = 3\} \qquad \{x = y\}\ x = x + y;\ \{x = 2y\}$$

Here, the symbol $=$ is overloaded to denote both the assignment statement in i and the equality symbol in P and Q.

When i is the entire body of a function, we call P a *precondition* for this function and Q a *postcondition* for this function. When the corresponding triple is valid, calling that function in a state that makes the precondition valid ensures that it ends in a state that makes the postcondition valid. When i is a loop body, we ask for P and Q to be identical, and we call it a *loop invariant*. When the corresponding triple is valid, entering the loop for the first time in a state that makes the loop invariant valid ensures that, upon exiting the loop, the same invariant is still valid. We are not considering here the problem of termination, which requires different annotations (e.g., *loop variant*) to ensure progress. We consider non-terminating computations as valid ones, except they may not exit from a loop or return from a function.

We are mostly interested in the automatic proof of the verification conditions produced by translating program assertions into Hoare triples. We do not detail here the inner working of automatic provers, like Simplify [13], Yices [15] or Ergo [11], that we will use as black boxes that answer "yes", "no" or "don't know" to queries about validity of formulas.

1.2.3 Common language

Since we aim at making AI and DV collaborate, we need a common language to communicate their results. The natural language to do so is first-order logic without quantifiers. We will see how quantifiers can be hidden in special variables that we describe in Section 3.3.

AI does not work directly with logical formulas. Instead, there is a natural translation, for each abstract domain we consider, from an abstract value of that domain to a quantifier-free formula. The abstract value and the formula represent the same set of concrete values. We can go further and consider the domain of equality and inequality formulas, like we do in Section 3.7. This is a kind of light predicate abstraction, that we can directly express in AI. We will use this equivalence of an abstract value and a first-order formula in Section 3.5, when we describe our method, to apply techniques of predicate calculus like Fourier-Motzkin elimination.

In the following, we will use the notation $\overline{a}$ for the negation of a, if a is a formula, or the complement of a, if a is a set of elements.

1.3 Context of interest

We developed our method together with an implementation in Caduceus, a tool dedicated to the verification of C programs. Caduceus accepts as input a subset of C excluding casts and unions, but with full support for pointers, dynamic allocation, structures and recursion. The memory model adopted for C is the Burstall-Bornat or component-as-array one [8, 6]. A JML-like language [25] allows programmers to annotate their C code with preconditions, postconditions, invariants and the like. This annotation language allows, in particular, to express logical properties not expressible in C, like validity of a pointer, so that Caduceus can check them. In our examples, these annotations will appear inside stylized C comments as in Figure 1.

```
//@ requires: precondition for function f
//@ ensures: postcondition for function f
void f() {
   ...
   //@ invariant: loop invariant
   while (...) {
      ...
   }
}
```

Figure 1: Logical annotations in Caduceus

Caduceus translates possibly annotated C code into an intermediate functional language that is fed into the verification condition generator WHY. WHY computes Hoare-style weakest preconditions and outputs verification conditions for a variety of automatic and interactive provers [**?**]. This process is sound only as much as the prover used. The main difficulty is the addition of annotations that describe, in the logic, the additional knowledge programmers might have about programs. Our work tends to free programmers from adding logical annotations which are seemingly rephrasing the code, e.g., the trivial

interval invariant on iterator i in the program excerpt of Figure 2, that depends on the also trivial precondition that precedes it.

```
//@ requires: 0 ≤ s
void loop(int s) {
   int i = 0;
   //@ invariant: 0 ≤ i ≤ s
   while (i++ < s) {
      ...
   }
}
```

Figure 2: Trivial annotations

1.4 Target language

We restrict our attention to a subset of C that exhibits the kind of constructs that usually make automatic verification impossible: pointer arithmetic and aliasing. This subset is kept very small intentionally for simplicity of exposition. We detail in Section 5 how we deal with more syntactic statements. The current limitations concern the treatment of casts, structures and memory deallocation:

casts are not supported at all in Caduceus;

structures and double indirection are supported by Caduceus, we just do not consider them in our memory safety analysis;

memory deallocation through calls to *free* is supported by the standard memory model of Caduceus, not by the modified memory model we use here.

Our small while-language, as defined in Figure 3, contains only two base types: integers and characters. Only one level of pointer is allowed, with no casts, which implicitly means that the value of pointer variables cannot be modified through aliasing. Only the memory location they point to can be modified through aliasing. Allocating memory is done by calling some typed C++-like *new* operator. String literals are supposed to be mutable memory blocks, like the memory allocated by calling *new*. Usual C operations are allowed on integers and pointers, like pointer arithmetic and dereferencing. Control flow statements are limited to *if* branching statements and *while* loops. Ansi C rules for syntax and semantics are adopted whenever applicable to this subset.

In C code samples, we will use == for equality testing and = for assignment, as commanded by Ansi C. In logical formulas, we will use = for equality testing, as there is no ambiguity with assignment there. Likewise, we will use C operators in C code samples (e.g., <=) and their logical notation in logical formulas (e.g., $\leq$). Finally, we define informally the following constructs that we use when defining our while-language.

type	::=	*int* | *char*	Base type
	|	*int* $\star$ | *char*$\star$	Pointer type
decl	::=	*type var*	Declaration
expr	::=	*lit*	Literal
	|	*var*	Variable
	|	*unop expr* | *expr unop*	Unary operation
	|	*expr binop expr*	Binary operation
	|	*expr assignop expr*	Assignment
	|	*func*(*expr**)	Function call
	|	*expr* [*expr*]	Indexed access
	|	*new int*[*expr*]	Int allocation
	|	*new char*[*expr*]	Char allocation
stat	::=	*expr*;	Simple
	|	*decl*; | *decl* = *expr*;	Declarations
	|	*stat stat* | { *stat* }	Compound
	|	*return expr*;	Return
	|	*lab* : *stat*	Label
	|	*if* (*expr*) *stat else stat*	Double branching
	|	*if* (*expr*) *stat*	Single branching
	|	*while* (*expr*) *stat*	Loop
fundef	::=	*type func*(*decl**) { *stat* }	Function
prog	::=	*fundef**	Program

Figure 3: While-language

lit is the set of literals (including integer, character and string literals),

var, *func* and *lab* are respectively the sets of variable, function and label names,

unop is the set of C unary operators: $\star$ + − ∼ ! ++ −− *sizeof*
The address-of operator & is not considered here, and this set includes operators not included in unary operators in the standard.

binop is the set of C binary operators:
$\star$ / % + − << >> < > <= >= == != & ∧ | && || ,
This set includes operators not included in binary operators in the standard.

assignop is the set of C assignment operators:
= $\star$= /= %= += −= <<= >>= &= ∧= |=

1.5 Method overview and motivating example

Section 2 will review the preliminary analyses needed for our modular and contextual method presented in Section 3 to work. Section 4 will detail our implementation of these analyses

and Section 5 the experiments we made to show our method's effectiveness. We will finish with related work in Section 6 and future work in Section 7. The example in Figure 4 will be used throughout this paper to explain each step of our analysis.

```
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   char⋆ cur = dest;
   if (dest == 0) return 0;
   while (s−− > 0 && ⋆src) {
      ⋆cur++ = ⋆src++;
   }
   ⋆cur = '\0';
   return dest;
}
```

Figure 4: Motivating example

The intended meaning of function *foo*, when *dest* is not null, is to copy at most s characters from string *src* to pointer *dest*, and then return *dest*. If *dest* is null, it returns null. It bears some resemblance with the standard string function *strncpy*, with differences too (e.g., in *foo*, if the length of string *src* is less than s, no zero padding is added).

2 Aliasing and strings

The modular and contextual analysis presented in Section 3 does not deal by itself with pointers, but mostly with integers and booleans. It relies on the preliminary analyses presented here to remove most pointer arithmetic and some local aliasing, explicit some local non-aliasing and identify strings. The analyses that follow are independent and could be applied on their own in other contexts. They classify each pointer value in the program (in its context of use) into:

a base pointer, a cursor pointer or a complex pointer

a string or a (plain) pointer

A *base pointer* at program label L is one of three kinds: the initial value of a pointer parameter, the result of a call to *new* on an execution path reaching L or the pointer value returned by a call to some function on an execution path reaching L.

A *cursor pointer* at program label L is a pointer value that can be shown to be always aliased with some other constant or variable offset expression from a same base pointer at L. We say the cursor pointer is *based* on the corresponding base pointer. E.g., if p is a base pointer, $p + 3$ and $p + f(q[i])$ are cursor pointers based on p. [1]

[1] C99 standard defines a similar notion of pointer *based* on an object when formally defining keyword *restrict* in §6.7.3.1., except it is a syntactic notion whereas ours is semantic.

A *complex pointer* at program label L is a pointer that is neither a base pointer nor a cursor pointer.

A *string* (or *string pointer*) at program label L is a pointer to an array of characters terminated by a sentinel, the null character.

A *(plain) pointer* at program label L is a pointer that is not a string. We will use the name *pointer* for *plain pointer* when it is clear from the context that we exclude strings.

2.1 Local aliasing

It follows from the definition of cursor pointers that their occurrences in the program could be replaced by pointer expressions only mentioning base pointers. As such, identifying as many cursor pointers as possible is profitable to verification, since it decreases the number of pairs of pointers that could be aliased.

A quick look at the example of Figure 4 shows this is possible even if the most precise aliasing information is not available: in function *foo*, pointer variable *cur* at any program point is a cursor pointer based on pointer parameter *dest*. At the point of dereferencing *cur* in the loop, for example, *cur* is aliased to $dest + i - 1$. Unfortunately, inferring this kind of aliasing information is not easy, it is the goal of the analysis presented in Section 3. What is much easier is to introduce an integer variable *cur_offset* that follows the value of $cur - dest$ in the program. We can then replace occurrences of *cur* with $dest + cur_offset$. This in turn allows us to discard totally pointer variable *cur*. The same can be done for occurrences of *src* in the program, which is a cursor pointer based on pointer parameter *src* (its own initial value). Then we can replace occurrences of *src* with $src + src_self_offset$ [2], on the condition that pointer arithmetic on *src* is replaced by integer arithmetic on src_self_offset. Although the definitions of base and cursor pointers do not apply to integer variables, the same transformation can be applied to s, which becomes $s + s_self_offset$. The program resulting from those transformations is presented in Figure 5.

The analysis needed for this program transformation can be formalized as AI over some special pointer domain, presented in Figure 6 with only two variables v_1 and v_2 and two integer constants c_0 and c_1, where elements point to their immediately greater element in the lattice ordering. In order to easily name base pointers, we introduce temporary variables to hold the result of calls to *new* or the result of calls to functions that return a pointer (as in static single assignment transformation). For the same reason, we introduce local variables to replace parameter variables inside the function, so that the name of a parameter denotes only the corresponding base pointer. We divide cursor pointers into index ones and offset ones. Index pointers are aliased to some constant offset expression from a base pointer. Offset pointers are aliased to some variable expression from a base pointer. This leads to the introduction of integer offset variables to follow the value of this difference from the current base pointer (which may not be the same at all program points).

[2] We use the name p_self_offset instead of p_offset when pointer p is a cursor pointer based on its own initial value, to increase readability.

```
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   while (s + s_self_offset−− > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 5: Example (cont.) : after local aliasing transformation

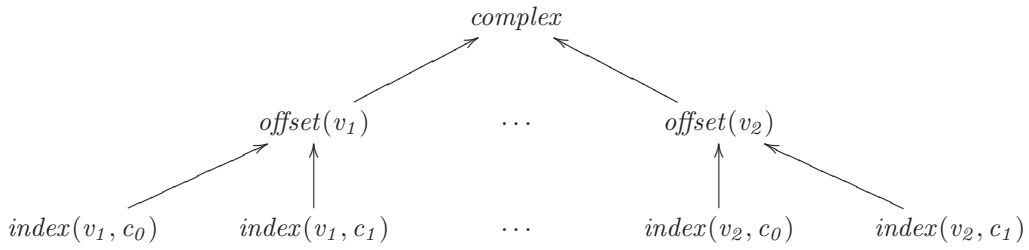

Figure 6: Lattice for local aliasing

The union and intersection operations are easily derived from the lattice ordering. Since the height of this lattice is bounded, we do not need a widening operator. Each program point is mapped to some abstract value in the point-wise lattice that maps each local pointer variable to a value in the local aliasing lattice. The transfer function for assignment is as expected: when assigning the pointer value v to pointer variable p, we compute the abstract value for v using the current abstract values of all local variables, and define this as the new abstract value of p.

Local aliasing transformation uses the results of this analysis to remove pointer arithmetic, local pointer aliasing and pointer variables that it replaces with integer arithmetic, equality on integers and integer variables. Before we transform the program at program point L, where pointer variable p is read or written, we need to take into consideration the abstract values associated to p not only at L but in all the program. Figure 7 shows an example where p is always *index* of some other pointer (either q or r) at program points it appears. This allows us to discard p altogether, and replace its value by the corresponding index on base pointer q or r where it is read. Figure 8 shows a slighly modified version of this function, where p is read with a *complex* abstract value for the last return statement. This makes it necessary to keep previous assignments to p.

```
int⋆ ok(int⋆ q, int⋆ r, int s) {
   int⋆ p = q;
   // p is index(q, 0)
   if (s > 0) {
      p = r;
      // p is index(r, 0)
      return p;
   }
   // p is complex
}
```

Figure 7: Transformation possible

```
int⋆ ko(int⋆ q, int⋆ r, int s) {
   int⋆ p = q;
   // p is index(q, 0)
   if (s > 0) {
      p = r;
      // p is index(r, 0)
   }
   // p is complex
   return p;
}
```

Figure 8: Transformation problem

There are various ways to solve this problem. We simply decided to lift all abstract values associated to p to the topmost abstract value kind in the lattice of Figure 6, among all such abstract values. Figure 9 describes this lifting in details. If needed, we could resort to a more precise solution. A first solution is to propagate backward the abstract values obtained by forward propagation. Iterated forward and backward propagations may lead to a better fixed point than the lifting just described. To go even further, we could separate the analysis of p's abstract value from the actual code needed to update p's value for future reads. This would make it unnecessary to change the abstract values computed by forward propagation. The somewhat odd result would be that p's value could be updated twice: once for writing p, once for writing p_offset.

topmost kind	*index*	*offset*	*complex*
old abstract value	$index(bp, c)$	$index(bp, _)$ or $offset(bp)$	_
new abstract value	$index(bp, c)$	$offset(bp)$	complex

Figure 9: Lifting abstract values

This lifting allows us to easily modify occurrences of p in the program. Given a language statement at program point L that reads or writes p, and depending on the abstract value of p at L, Figure 10 describes the transformation performed on the code. We note $[.]_{offset}$ the interpretation of a pointer expression as an offset. We are guaranteed that such an interpretation exists when we use it in Figure 10 since the local aliasing analysis computed an abstract value of $offset(bp)$ for p at that point. Figure 11 gives a partial definition for this interpretation. If originally present in the program, logical annotations too have to be translated, which only adds minor practical difficulties. The last step of this transformation consists in removing the definition of pointers that were lifted as *index* or *offset*, and to introduce instead definitions for the offset variables used.

language statement	p	p = e
p is $index(bp, c)$	bp + c	e
p is $offset(bp)$	bp + p_offset	p_offset = $[e]_{offset}$
p is *complex*	p	p = e

Figure 10: Lifting abstract values

$$
\begin{array}{rcll}
[p]_{offset} & = & c & \text{if p is } index(bp, c) \\
[p]_{offset} & = & p_offset & \text{if p is } offset(bp) \\
[e_1 + e_2]_{offset} & = & [e_1]_{offset} + e_2 & \text{if } e_1 \text{ is a pointer expression} \\
[e_1 - e_2]_{offset} & = & [e_1]_{offset} - e_2 & \text{if } e_1 \text{ is a pointer expression}
\end{array}
$$

Figure 11: Partial definition of $[.]_{offset}$

This transformation allows us to remove all the simple pointer arithmetic and some local aliasing, possibly all the local aliasing introduced by local variables in a function. The same analysis can also be used to rewrite pointer comparisons (resp. pointer differences) as integer ones when comparing (resp. subtracting) cursor pointers with the same base pointer (or a cursor pointer with its base pointer). This is a simple but crucial step, since it is common practice in C to use pointer arithmetic when iterating over an array for efficiency purposes. When initialization is not a concern, either because a static analysis can guarantee that all reads of pointer values occur after proper initialization, or because it creates the corresponding verification conditions to be checked by DV, then we can improve precision by adding a least element *uninitialized* to this lattice. We do not detail it here, but this technique can also be applied on integer expressions to help discover relational invariants. This is what we did when modifying occurrences of s in Figure 5. Figures 12 and 13 show code samples before and after local aliasing transformation.

2.2 Local non-aliasing

Section 2.1 did not deal with aliasing between different base pointers. This kind of aliasing cannot in general be inferred locally, but according to the following idiom, *non-aliasing* of base pointers can be locally inferred.

Idiom 1 *Given two different base pointers p and q, a memory location written through some pointer expression based on p must not be consequently read through some pointer expression based on q.*

This idiom expresses the local uniqueness of names to access modified memory locations. These names may not be unique in the original program, as far as they all refer to the same base pointer. It facilitates not only automated reasoning about the program but also human reasoning, which makes it important to follow in practice. The easiest way to guarantee that this idiom is respected when base pointer p is read through after base pointer q is

```
int f(int);

int* g(int* p, int n, int d) {

   int* r;

   if (p == 0) {
      p = new int[n];

   }
   p += d;
   while (n--) {
      *p++ = 0;
      if (f(*p)) {
         r = p;
      }
   }
   return r;
}

char* h(char* p, int s) {


   char* q;
   if (s > 0) {
      q = p+s;
      while (p < q) {
         p +=
            f(q-p);
      }
      return q;
   } else {
      q = new char[s];

      while (f(*q) > 0) {
         *q++ = '\0';
      }
      return q;
   }
}
```

Figure 12: Original program

```
int f(int);

int* g(int* p, int n, int d) {
   int p_self_offset = 0;
   int n_self_offset;
   int r_offset;
   if (p == 0) {
      p = new int[n];
      p_self_offset = 0;
   }
   p_self_offset = p_self_offset + d;
   while (n + n_self_offset--) {
      p[p_self_offset++] = 0;
      if (f(p[p_self_offset])) {
         r_offset = p_self_offset;
      }
   }
   return p + r_offset;
}

char* h(char* p, int s) {
   int p_self_offset = 0;
   int q_offset;
   char* q;
   if (s > 0) {
      q_offset = p_self_offset + s;
      while (p_self_offset < q_offset) {
         p_self_offset +=
            f(q_offset - p_self_offset);
      }
      return p + q_offset;
   } else {
      q = new char[s];
      q_offset = 0;
      while (f(q[q_offset]) > 0) {
         q[q_offset++] = '\0';
      }
      return q + q_offset;
   }
}
```

Figure 13: After transformation

written through is to ask for p and q to be *separated*. The need for annotations on pointer separation in a modular setting has been noted previously in [24]. Our contribution is to allow automatic inference of these annotations, using idiom 1. We introduce three related predicates to express this non-aliasing property:

> $separated(p, q)$ expresses that pointers p and q do not point to the same memory location. This is not the same as asking for $p \neq q$, since p and q may be equal as far as they do not point to some memory location. An important such case is when p and q are null.
>
> $full_separated(p, q)$ expresses that pointers p and q do not point to the same memory block. This is stronger than asking for simple separation. In our memory model, memory blocks allocated through different dynamic calls to some allocation function are not reachable one from the other. This makes the full separation of p and q equivalent to the separation of $p + i$ and $q + j$ for any integers i and j :
>
> $$full_separated(p, q) \equiv \forall\ int\ i, j.\ separated(p + i, q + j) \tag{7}$$
>
> $bound_separated(p, n, q, m)$ expresses that the memory chunks deliminated by p included and $p+n$ excluded on one side, q included and $q+m$ excluded on the other side, do not overloap. This is weaker than asking for full separation. Simple separation can be seen as a special case of bounded separation:
>
> $$separated(p, q) \equiv bound_separated(p, 1, q, 1) \tag{8}$$

These relations between predicates can be formally stated as the lattice of Figure 14, where p and q are pointer variables and i, j, k and l are integers (not necessarily integer constants). In our analysis, such an integer i represents a constant or symbolic range $[0..i[$ of indices at which p is read or written in the function.

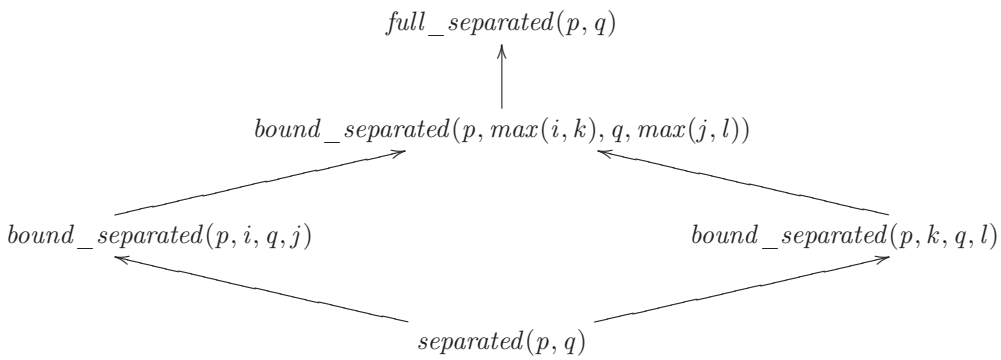

Figure 14: Lattice of separation

Asking for any of these properties of base pointers can be formalized as a precondition or postcondition of functions, depending on the kind of base pointers involved. Figure 15 shows how these requirements can be translated into function precondition or postcondition depending on the kind of base pointers involved in the separation predicate. Here sep-pred is any of *separated*, *bound_separated* or *full_separated*.

sep-pred(p,q)	p is parameter	p is result of *new*	p is result of call
q is parameter	precondition	true	postcondition
q is result of *new*	true	true	postcondition
q is result of call	postcondition	postcondition	postcondition

Figure 15: Separation preconditions and postconditions

We put true in this table whenever the separation condition holds by construction in our memory model, which is the case between pointers obtained by two different calls to *new*, or such a pointer and a parameter of the function. Separation between two parameters easily translates to a precondition of the function. Separation with a pointer obtained as the result of a call is more complex. It involves knowing how this result was obtained, more precisely we need to know whether this result can be pointing to the same memory block as the arguments used for the call. Depending on this information, the separation condition can be true, or expressed as a postcondition on the called function, which can sometimes translate to a precondition for the call.

This defines a transfer function on the call-graph, to be used in AI over the lattice of Figure 14, generating preconditions and postconditions of functions along the way. On our motivating example, this results in a precondition for function *foo*, namely $full_separated(dest, src)$, that amounts to asking for the non-overlapping of *src* reads and *dest* writes in the loop, which is the expected behaviour. Figures 17 and 18 show code samples where the preconditions are respectively ideal separation conditions based on idiom 1 and the actual separation conditions inferred by our analysis.

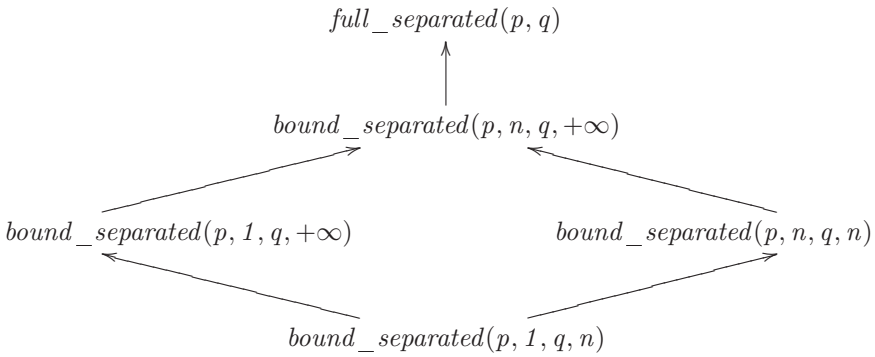

Figure 16: Possible preconditions for *copy*

```
// no precondition
void select(int* r, int* p, int* q) {
   if (*p < *q) *r = *p;
   else *r = *q;
}

//@ requires: bound_separated(p, 1, q, n)
void copy(int* p, int* q, int n) {
   int i = 0;
   while (i++ < n) *p++ = *q++;
}

//@ requires: separated(p, q)
void min(int* p, int* q, int n) {
   int i = 0;
   while (++i < n) {
      select(q, q, q+i);
   }
   copy(p, q, 1);
}
```

Figure 17: Ideal separation

```
// no precondition
void select(int* r, int* p, int* q) {
   if (*p < *q) *r = *p;
   else *r = *q;
}

//@ requires: full_separated(p, q)
void copy(int* p, int* q, int n) {
   int i = 0;
   while (i++ < n) *p++ = *q++;
}

//@ requires: full_separated(p, q)
void min(int* p, int* q, int n) {
   int i = 0;
   while (++i < n) {
      select(q, q, q+i);
   }
   copy(p, q, 1);
}
```

Figure 18: Actual separation

There is still room for improvements, in various directions. Figure 16 shows the ordering between the actual precondition we find for function *copy*, the ideal precondition we would like to have, and various intermediate preconditions that would improve on the current one.

2.3 Pointers and strings

Plain pointers and strings are used in quite different ways in programs. Knowing which base pointers are strings and which are not allows us to infer useful information for each. Being a string is not a type information in C, since direct access to the string representation allows (re)moving the null sentinel character. This is more a typestate [34]. While a type is a predicate that characterizes an object throughout the program, a typestate is a predicate that characterizes an object at a precise location in the program. First of all, only pointers to characters are eligible for being strings. Otherwise, we approximate the string typestate using the following idioms.

Idiom 2 *Some arguments of some library functions calls are strings (e.g., both arguments of calls to* `strcat`*).*

Idiom 3 *Character pointers whose value at some index is tested for nullity are strings.*

These heuristics give us the seed information that we propagate backward to infer string preconditions and postconditions on function parameters and function returns. We define

a new predicate *string* for that purpose. Although the backward propagation is not sound, which is acceptable for an inference method, the forward propagation that follows it has to take into account the separation information to keep string information up-to-date. Overall, this back-and-forth method is a particular case of the method we describe in Section 3. We will detail its specificities after the general method has been presented. A crucial point here is that inference is based on backward propagation only, which is the key feature to ensure modularity.

3 Modular and contextual verification

We designed a very precise intra-procedural analysis, that is both flow-sensitive and path-sensitive in order to capture the possible complex data dependences that account for memory safety. To make it scalable despite its high local precision, we adopted a modular framework to communicate preconditions and postconditions of functions throughout the call-graph. Finally, our analysis is not so much context-sensitive as contextual: the context of interest for analysing a function is not defined by the calling contexts (context-sensitive analysis) but by need, according to the function's body sensitivity to context (contextual analysis).

We consider here that the preliminary analyses dealing with aliasing described in Section 2 have been run, so that functions have appropriate preconditions and postconditions on pointer separation. Also, most pointer arithmetic and local aliasing should have been transformed into integer manipulations. This leads to the intermediate annotated code shown in Figure 20 for our running example. We will explain later how string information is computed, based on the general analysis we present here. Figure 19 presents our method schematically.

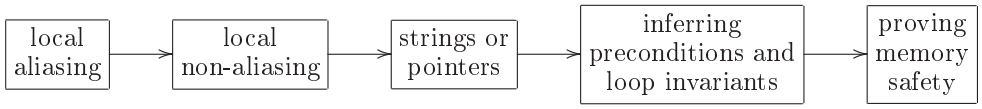

Figure 19: Schematic view of our method

3.1 Verification frameworks

We intend to infer necessary invariants based on Abstract Interpretation (AI), and prove memory safety either directly by Abstract Interpretation or by generating verification conditions for Deductive Verification (DV). In an optimized mode, we would like to prove memory safety using AI whenever possible, and otherwise resort to DV. In a safer mode, we would prefer to use AI only for inferring invariants and preconditions, and leave all proofs to DV. In the following, we will refer to these verification frameworks as:

AI×DV : invariant generation by AI, proof by AI or DV

```
//@ requires: full_separated(dest, src)
char* foo(char* dest, char* src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   while (s + s_self_offset-- > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 20: Example (cont.): before modular and contextual analysis

AI+DV : invariant generation by AI, proof by DV only

In either framework, communication between AI and DV will be limited to loop invariants and function preconditions, as well as the set of verification conditions proved by AI in the context of AI×DV.

Another kind of collaboration between AI and DV has been explored in [26], where invariants inferred by AI as well as the counter-example from DV are given to DV, which returns if possible a refined loop invariant. This approach suffers a drawback that makes it difficult to use in practice and that our method avoids. It requires using DV repeatedly, which is too costly for being used on real programs.

3.2 Memory model

We build on the memory model of Caduceus [16], which follows the Burstall-Bornat or component-as-array one [8, 6]. It represents the memory by a finite set of variables, each containing an applicative map. We do not detail it more, as we rely only on very simple assumptions on this memory model, since we do not consider structures and casts. Each pointer value has an associated block that corresponds to the result of the initial call to some allocation function. Memory blocks allocated through different dynamic calls to some allocation function are not reachable one from the other.

We first define a fresh base pointer p at runtime to be a base pointer that is fully separated from all previously known base pointers q, so that $full_separated(p, q)$ holds. In the small while-language described in Figure 3 that we consider here, there are only two ways to create a fresh base pointer: either by using a string literal, which creates a string pointer, or by calling operator *new* to allocate a block of memory, which creates a plain pointer. There is no way to deallocate a block of memory, which allows us to treat the length of a block of memory as a constant. String literals are treated much like initialized blocks of memory allocated with operator *new*, ended by a trailing null character.

To formalize these informations about fresh base pointers, we introduce logical partial functions *arrlen* and *strlen*. On the AI side, the absence of quantification makes it costly to apply these partial functions to all pointers, therefore we define them only on base pointers. On the DV side, they are potentially defined for all pointers through quantification. It is the axiomatization that accurately gives their domain of definition.

> $arrlen(p)$ is the length of the array pointed-to by pointer p. It is potentially defined for any pointer p.
>
> $strlen(p)$ is (not surprisingly) the length of the string pointed-to by p. It is potentially defined only for strings, i.e. for pointers to null-terminated character arrays.

The following formulas hold for fresh base pointers, with n and m integer expressions that give respectively the size of the memory block allocated by *new* and the size of the memory occupied by a string, including the trailing null character.

$$arrlen(p) = n \tag{9}$$

$$string(p) \rightarrow strlen(p) = m - 1 \tag{10}$$

Given a plain pointer or a string p, it is now possible to express the safety of a read or write access to p at some index i as validity of a logical formula based on *arrlen* or *strlen*. To get simple and useful formulas, we rely on the following idioms.

Idiom 4 *A base pointer can be only read and written through at positive indices.*

Idiom 5 *A string can only be read up to its sentinel null character.*

Idiom 4 is trivially true for fresh base pointers, that point by construction to the beginning of a memory block. After the local aliasing transformation described in Section 2.1, we expect it to be true of all base pointers in most programs. Actually, this is mostly a convenient idiom to simplify our implementation and facilitate the work of automatic provers. There is no theoretical difficulty in adding a negative array length function. Idiom 5 is generally respected for all unbounded data structures terminated by a sentinel, in particular for strings. This leads to the following formulas for expressing the safety of a read access $p[i]$, with p a pointer and i an integer expression.

p is a	safe read
plain pointer	$0 \leq i < arrlen(p)$
string	$0 \leq i \leq strlen(p)$

Figure 21: Safety condition for a read access

Writing to a string is more complex, since it may destroy the very fact it is a string. To increase the precision of our analysis, we take into account the expected typestate of

p is a	after write, p is a	v is null	safe write
pointer	-	-	$0 \leq i < arrlen(p)$
string	pointer	-	$0 \leq i \leq strlen(p)$
string	string	yes	$0 \leq i \leq strlen(p)$
string	string	-	$0 \leq i < strlen(p)$

Figure 22: Safety condition for a write access

the pointer after the write access and the nullity of the value written. This leads to the following formulas for expressing the safety of a write access $p[i]$, with p a pointer, i an integer expression, and v the value written.

On the AI side, $arrlen(p)$ and $strlen(p)$ are treated like meta-variables, whose value changes at runtime. Transfer functions take those special variables into account with their specificities. In particular, any information on $strlen(p)$ implicitly carries the extra information that p is a string, i.e. that $string(p)$ holds. Conversely, being a string can be expressed in terms of $strlen(p)$.

$$string(p) \rightarrow 0 \leq strlen(p) \tag{11}$$

Conversions from plain pointer to string and back are reflected on the AI side between $arrlen(p)$ and $strlen(p)$. When writing a null character into a plain pointer p at positive index i, with p expected to be a string after the assignment, the forward transfer function for this assignment adds constraints $strlen(p) \leq i$ and $strlen(p) < arrlen(p)$ to the current abstract state. Conversely, when writing into a string p at unbounded indices, e.g., in a loop, we loose the information that p is a string, which translates on the AI side by adding the constraint $strlen(p) < arrlen(p)$ before removing information on $strlen(p)$. AI knowledge of *arrlen* and *strlen* semantics is otherwise limited to the basic equations that follow.

$$arrlen(0) = 0 \tag{12}$$

$$string(p) \rightarrow 0 \leq strlen(p) < arrlen(p) \tag{13}$$

On the DV side, *arrlen* and *strlen* are treated like uninterpreted functions, for which an appropriate axiomatization is given. We give here a simplified signature for these functions.

$$arrlen : \alpha\ pointer \rightarrow int$$

$$strlen : \alpha\ memory \times \alpha\ pointer \rightarrow int$$

The type parameter α is used to make *arrlen* and *strlen* polymorphic in the type of pointer they take as argument. The function *strlen* has an additional parameter that represents the memory state at the point it is called, in order to take into account the possibility of string manipulations. This extra parameter with respect to AI is set up automatically by the tool Why in which our plugin works, during the computation of weakest preconditions. If p is a pointer, i an integer expression, m the memory state at the program point we consider,

C expression	$p+i$	$*p=v$	$*p$
logical expression	$shift(p,i)$	$upd(m,p,v)$	$acc(m,p)$

Figure 23: Correspondance between C and logical expressions

v a value, we can write the correspondances given in Figure 23 between C expressions and their logical counterpart.

The axiomatization that follows defines the basic properties of *arrlen* and *strlen*. It uses the existing logical functions *shift*, *upd* and *acc*. Currently, lemmas derived from these axioms are also given as axioms in order to facilitate the task of DV. All variables that appear in these axioms are universally quantified. We do not show the prefix of the prenex form here to increase readability.

Reading or writing through a pointer p does not modify $arrlen(p)$. Only the effect of pointer arithmetic must be defined on $arrlen(p)$.

$$0 \leq i \rightarrow arrlen(shift(p,i)) = arrlen(p) - i$$

The evident relation between the length of a string and the size of the memory block for this string must be given.

$$0 \leq strlen(m,p) \rightarrow strlen(m,p) < arrlen(p)$$

The effect of pointer arithmetic on $strlen(m,p)$ is the same as on $arrlen(p)$.

$$0 \leq i \rightarrow strlen(m, shift(p,i)) = strlen(m,p) - i$$

Reading a string inside its bounds returns a non-null character. Reading it at its length index returns the sentinel null character.

$$0 \leq i < strlen(m,p) \rightarrow acc(m, shift(p,i)) \neq 0$$
$$0 \leq strlen(m,p) \rightarrow acc(m, shift(p, strlen(m,p))) = 0$$

As expected, writing a string is the more complex operation. We consider first writing a character in the string bounds.

$$0 \leq i < strlen(m,p) \rightarrow strlen(upd(m, shift(p,i), v), p) \leq strlen(m,p)$$

We can be more precise if we know the value written is null or the opposite.

$$0 \leq i \leq strlen(m,p) \rightarrow strlen(upd(m, shift(p,i), 0), p) = i$$
$$0 \leq i < strlen(m,p) \wedge v \neq 0 \rightarrow strlen(upd(m, shift(p,i), v), p) = strlen(m,p)$$

Writing a null character may construct a string.

$$0 \leq i \rightarrow 0 \leq strlen(upd(m, shift(p,i)), 0) \leq i$$

If we know all characters are not null in the interval $[0..i[$, then we can give the actual length of the string created.

$$0 \leq i \ \wedge \ (0 \leq j < i \rightarrow acc(m, shift(p, j)) \neq 0) \rightarrow strlen(upd(m, shift(p, i), 0), p) = i$$

It can come to a surprise that *strlen* is axiomatized instead of being implicitly defined, like this:

$$\begin{aligned} string(m, p) \equiv \exists\ int\ i\ .\quad & i = strlen(m, p) \\ & \wedge\ acc(m, shift(p, i)) = 0 \\ & \wedge\ \forall\ int\ j.\ 0 \leq j < i \rightarrow acc(m, shift(p, i)) \neq 0 \end{aligned}$$

Early experiments with that formulation showed it was not adapted to DV, mostly because of the presence of existential quantification. The interest of using *strlen* with axioms is to hide this quantifier from DV.

It can be seen from these axioms that *strlen* has quite a different meaning in DV than in AI. While the emphasis is on idioms in AI, so that the inferred invariants are likely to be invariants meant by the programmers, emphasis in DV is on completeness. The expression of these axioms as linear inequalities is particularly well suited for using automatic DV, as we will see in Section 5 when we compare with an alternate formulation. These experiments with various memory models were only possible because Caduceus provides a very parametric basic memory model.

3.3 Meta-variables and uninterpreted functions

So far, we have seen that *arrlen* and *strlen*, that express respectively the size of a memory block and the length of a string, can be used as meta-variables generators in AI and as uninterpreted functions in DV. This is a very useful trick that has been examplified by [10], where meta-variables serve as designator shortcuts for expressions in the program and where they are used mostly as black boxes whose validity only is checked by AI. We go a step further in two directions with *arrlen* and *strlen*, since we introduce for the purpose of verification extra expressions not mentioned in the program text, and we specialize transfer functions in AI for these meta-variables.

We now introduce other such meta-variables generators: *min* and *max*. As expected from their name, $min(a, b)$ represents the minimum of integer values a and b, while $max(a, b)$ represents their maximum. Contrary to *arrlen* and *strlen* that only apply to the pointers present in the program text, *min* and *max* could be used to generate an infinity of meta-variables: $min(a, b)$ but also $min(strlen(p), a)$ and $min(min(strlen(p), a), b)$, etc. The way we introduce such variables in our analysis prevents us from generating an infinity of these variables.

We will see in Section 3.5 why we need *min* and *max*. Their use in AI is derived from their role as lower and upper bounds: when introducing $min(a, b)$ or $max(a, b)$, we also add the following constraints to the current abstract value:

$$min(a, b) \leq a \wedge min(a, b) \leq b$$

$$max(a,b) \geq a \wedge max(a,b) \geq b$$

The corresponding axioms are used in DV, we do not detail them here.

3.4 Implication lattice

Before we get into the back-and-forth propagation that generates invariants from memory safety conditions, we need to introduce a new lattice for propagating implications like $\psi \rightarrow \phi$ through the code.

Consider two abstract complete lattices A and B, with appropriate union and intersection operations, that we denote respectively $\sqcup$ and $\sqcap$. We assume there is a Galois connection from the set Φ of first-order logic formulas without quantifiers to both A and B (as in [10]), where abstraction and concretization functions are denoted respectively *aval* and *pred*. An implication lattice $A \Rightarrow B$ of A and B is a lattice whose carrier $C_{A\Rightarrow B}$ is a subset of $A \times B$ such that any pair (a,b) represents exactly the logical implication of the concretizations of a and b. By definition, the following relation holds:

$$pred_{A\Rightarrow B}(a,b) \triangleq pred_A(a) \rightarrow pred_B(b). \tag{14}$$

To allow more efficient implementations, we do not require that the carrier is the full set $A \times B$. Instead, we extend the implication lattice over $A \times B$ by mapping any pair (a,b) to a representative denoted $a \rightharpoonup b$ in the carrier such that:

$$pred_A(a) \rightarrow pred_B(b) \text{ logically implies } pred_{A\Rightarrow B}(a \rightharpoonup b). \tag{15}$$

This mapping is the identity on representatives, so that we identify (a,b) and $a \rightharpoonup b$ on $C_{A\Rightarrow B}$. Take now $a_1 \rightharpoonup b_1$ and $a_2 \rightharpoonup b_2$ from $C_{A\Rightarrow B}$. We define a union and an intersection operations over $A \Rightarrow B$ as follows:

$$(a1 \rightharpoonup b1) \sqcup_{A\Rightarrow B} (a2 \rightharpoonup b2) \triangleq (a1 \sqcap_A a2) \rightharpoonup (b1 \sqcup_B b2) \tag{16}$$

$$(a1 \rightharpoonup b1) \sqcap_{A\Rightarrow B} (a2 \rightharpoonup b2) \triangleq (a1 \sqcup_A a2) \rightharpoonup (b1 \sqcap_B b2) \tag{17}$$

To ensure intersection correctly under-approximates concrete intersection in the cases where only relation 15 holds, we still ask that equation 14 holds for special abstract values:

$$pred_{A\Rightarrow B}((a1 \sqcup_A a2) \rightharpoonup (b1 \sqcap_B b2)) = pred_A(a1 \sqcup_A a2) \rightarrow pred_B(b1 \sqcap_B b2) \tag{18}$$

We define a natural ordering on $C_{A\Rightarrow B}$:

$$(a1 \rightharpoonup b1) \leq_{A\Rightarrow B} (a2 \rightharpoonup b2) \triangleq a2 \leq_A a1 \wedge b1 \leq_B b2, \tag{19}$$

and require that the extension to $A \times B$ respects this ordering:

$$a2 \leq_A a1 \wedge b1 \leq_B b2 \text{ logically implies } (a1 \rightharpoonup b1) \leq_{A\Rightarrow B} (a2 \rightharpoonup b2). \tag{20}$$

Least and greatest elements are now easy to define:

$$\bot_{A\Rightarrow B} \triangleq \top_A \rightharpoonup \bot_B \qquad\qquad \top_{A\Rightarrow B} \triangleq \bot_A \rightharpoonup \top_B$$

Theorem 1 $(A \Rightarrow B, \leq_{A\Rightarrow B}, \bot_{A\Rightarrow B}, \top_{A\Rightarrow B}, \sqcup_{A\Rightarrow B}, \sqcap_{A\Rightarrow B})$ *forms a complete lattice.*

Proof. We first show that $\sqcup_{A\Rightarrow B}$ and $\sqcap_{A\Rightarrow B}$ are compatible with the ordering relation $\leq_{A\Rightarrow B}$. Take $a1 \rightharpoonup b1$ and $a2 \rightharpoonup b2$ such that $(a1 \rightharpoonup b1) \leq_{A\Rightarrow B} (a2 \rightharpoonup b2)$.

$$\begin{array}{lll} & (a1 \rightharpoonup b1) \sqcup_{A\Rightarrow B} (a2 \rightharpoonup b2) & \\ = & (a1 \sqcap_A a2) \rightharpoonup (b1 \sqcup_B b2) & \text{by (16)} \\ = & a2 \rightharpoonup b2 & \text{by (19)} \end{array}$$

and

$$\begin{array}{lll} & (a1 \rightharpoonup b1) \sqcap_{A\Rightarrow B} (a2 \rightharpoonup b2) & \\ = & (a1 \sqcup_A a2) \rightharpoonup (b1 \sqcap_B b2) & \text{by (17)} \\ = & a1 \rightharpoonup b1 & \text{by (19)} \end{array}$$

From equation 20, we immediately get

$$\begin{array}{l} a1 \rightharpoonup b1 \leq_{A\Rightarrow B} (a1 \rightharpoonup b1) \sqcup_{A\Rightarrow B} (a2 \rightharpoonup b2), \\ a2 \rightharpoonup b2 \leq_{A\Rightarrow B} (a1 \rightharpoonup b1) \sqcup_{A\Rightarrow B} (a2 \rightharpoonup b2), \\ (a1 \rightharpoonup b1) \sqcap_{A\Rightarrow B} (a2 \rightharpoonup b2) \leq_{A\Rightarrow B} a1 \rightharpoonup b1, \\ (a1 \rightharpoonup b1) \sqcap_{A\Rightarrow B} (a2 \rightharpoonup b2) \leq_{A\Rightarrow B} a2 \rightharpoonup b2. \end{array}$$

This proves the compatibility of $\sqcup_{A\Rightarrow B}$ and $\sqcap_{A\Rightarrow B}$ with the ordering relation $\leq_{A\Rightarrow B}$. The existence of a least and greatest elements of any set of elements in $A \Rightarrow B$ is a direct consequence of the definitions of $\sqcup_{A\Rightarrow B}$ and $\sqcap_{A\Rightarrow B}$, given that A and B are themselves complete lattices. By definition of $\bot_{A\Rightarrow B}$, $\top_{A\Rightarrow B}$, those are the correct bottom anf top elements of this lattice. □

We assume there is a canonical monotone implication form $\psi \rightarrow \phi$ for any formula f, such that:

$$\psi = pred_A \circ aval_A(\psi). \tag{21}$$

A trivial such formula is $True \rightarrow f$. In particular, we assume every implication in the image of $pred_{A\Rightarrow B}$ is of this form. We also assume that Galois connection on A is in fact a Galois insertion:

$$a = aval_A \circ pred_A(a). \tag{22}$$

Abstraction function $aval_{A\Rightarrow B}$ is defined by:

$$aval_{A\Rightarrow B}(\psi \rightarrow \phi) \triangleq aval_A(\psi) \rightharpoonup aval_B(\phi). \tag{23}$$

Theorem 2 $(aval_{A\Rightarrow B}, pred_{A\Rightarrow B})$ *defines a valid Galois connection between* Φ *and* $A \Rightarrow B$.

Proof. By definition, $aval_{A\Rightarrow B}$ and $pred_{A\Rightarrow B}$ are monotone functions. First we prove that $pred_{A\Rightarrow B} \circ aval_{A\Rightarrow B}$ over-approximates identity.

$$
\begin{array}{lll}
 & pred_{A\Rightarrow B} \circ aval_{A\Rightarrow B}(\psi \rightarrow \phi) & \\
= & pred_{A\Rightarrow B}(aval_A(\psi) \rightharpoonup aval_B(\phi)) & \text{by (23)} \\
\Leftarrow & pred_A \circ aval_A(\psi) \rightarrow pred_B \circ aval_B(\phi) & \text{by (15)} \\
\Leftarrow & pred_A \circ aval_A(\psi) \rightarrow \phi & \text{by (4)} \\
= & \psi \rightarrow \phi & \text{by (21)}
\end{array}
$$

Next we prove that $aval_{A\Rightarrow B} \circ pred_{A\Rightarrow B}$ under-approximates identity. To that end, we first notice that $aval_B \circ pred_B(b) \leq_B b$, since $(aval_B, pred_B)$ is a Galois connection. Therefore by 19:

$$(a \rightharpoonup aval_B \circ pred_B(b)) \leq_{A\Rightarrow B} (a \rightharpoonup b) \tag{24}$$

Therefore:

$$
\begin{array}{lll}
 & aval_{A\Rightarrow B} \circ pred_{A\Rightarrow B}(a \rightharpoonup b) & \\
= & aval_{A\Rightarrow B}(pred_A(a) \rightarrow pred_B(b)) & \text{by (14)} \\
= & aval_A \circ pred_A(a) \rightharpoonup aval_B \circ pred_B(b) & \text{by (23)} \\
= & a \rightharpoonup aval_B \circ pred_B(b) & \text{by (22)} \\
\leq_{A\Rightarrow B} & a \rightharpoonup b & \text{by (24)}
\end{array}
$$

To be able to compute in the abstract, we must also prove that $\sqcup_{A\Rightarrow B}$ over-approximates the concrete union and $\sqcap_{A\Rightarrow B}$ under-approximates the concrete intersection. We first prove that $\sqcup_{A\Rightarrow B}$ over-approximates the concrete union.

$$
\begin{array}{lll}
 & pred_{A\Rightarrow B}((a1 \rightharpoonup b1) \sqcup_{A\Rightarrow B} (a2 \rightharpoonup b2)) & \\
= & pred_{A\Rightarrow B}((a1 \sqcap_A a2) \rightharpoonup (b1 \sqcup_B b2)) & \text{by (16)} \\
\Leftarrow & pred_A(a1 \sqcap_A a2) \rightarrow pred_B(b1 \sqcup_B b2) & \text{by (15)} \\
\Leftarrow & (pred_A(a1) \wedge pred_A(a2)) \rightarrow (pred_B(b1) \vee pred_B(b2)) & \text{by (2) and (3)} \\
= & (pred_A(a1) \rightarrow pred_B(b1)) \vee (pred_A(a2) \rightarrow pred_B(b2)) & \text{by de Morgan} \\
= & pred_{A\Rightarrow B}(a1 \rightharpoonup b1) \vee pred_{A\Rightarrow B}(a2 \rightharpoonup b2) & \text{by (14)}
\end{array}
$$

We also prove that $\sqcap_{A\Rightarrow B}$ under-approximates the concrete intersection.

$$
\begin{array}{lll}
 & pred_{A\Rightarrow B}((a1 \rightharpoonup b1) \sqcap_{A\Rightarrow B} (a2 \rightharpoonup b2)) & \\
= & pred_{A\Rightarrow B}((a1 \sqcup_A a2) \rightharpoonup (b1 \sqcap_B b2)) & \text{by (17)} \\
= & pred_A(a1 \sqcup_A a2) \rightarrow pred_B(b1 \sqcap_B b2) & \text{by (18)} \\
\Rightarrow & (pred_A(a1) \vee pred_A(a2)) \rightarrow (pred_B(b1) \wedge pred_B(b2)) & \text{by (2) and (3)} \\
= & \overline{pred_A(a1)} \wedge \overline{pred_A(a2)} \vee pred_B(b1) \wedge pred_B(b2) & \text{by de Morgan} \\
\Rightarrow & \overline{pred_A(a1)} \wedge \overline{pred_A(a2)} \vee pred_B(b1) \wedge \overline{pred_B(b2)} & \text{by de Morgan} \\
 & \vee \overline{pred_A(a1)} \wedge pred_B(b2) \vee pred_B(b1) \wedge \overline{pred_A(a2)} & \\
= & (pred_A(a1) \rightarrow pred_B(b1)) \wedge (pred_A(a2) \rightarrow pred_B(b2)) & \text{by de Morgan} \\
= & pred_{A\Rightarrow B}(a1 \rightharpoonup b1) \wedge pred_{A\Rightarrow B}(a2 \rightharpoonup b2) & \text{by (14)}
\end{array}
$$

□

If B is stable by intersection (concretization of intersection is intersection of concretization), e.g., for the convex lattices we use most often in practice, we can also give an over-approximation $\Cap_{A\Rightarrow B}$ of the concrete intersection:

$$(a1 \rightharpoonup b1) \Cap_{A\Rightarrow B} (a2 \rightharpoonup b2) \triangleq (a1 \sqcap_A a2) \rightharpoonup (b1 \sqcap_B b2) \tag{25}$$

This operation allows strengthening invariants, which makes it generally more useful than the intersection operation. We can prove it correctly over-approximates the concrete intersection:

$$
\begin{array}{lll}
 & pred_{A\Rightarrow B}((a1 \to b1) \Cap_{A\Rightarrow B} (a2 \to b2)) & \\
= & pred_{A\Rightarrow B}((a1 \sqcap_A a2) \to (b1 \sqcap_B b2)) & \text{by (25)} \\
\Leftarrow & pred_A(a1 \sqcap_A a2) \to pred_B(b1 \sqcap_B b2) & \text{by (15)} \\
\Leftarrow & (pred_A(a1) \wedge pred_A(a2)) \to (pred_B(b1) \wedge pred_B(b2)) & \text{by (2) and (5)} \\
= & \overline{pred_A(a1)} \vee \overline{pred_A(a2)} \vee pred_B(b1) \wedge pred_B(b2) & \text{by de Morgan} \\
\Leftarrow & \overline{pred_A(a1)} \wedge \overline{pred_A(a2)} \vee pred_B(b1) \wedge \overline{pred_B(b2)} & \text{by de Morgan} \\
 & \vee \overline{pred_A(a1)} \wedge pred_B(b2) \vee pred_B(b1) \wedge \overline{pred_A(a2)} & \\
= & (pred_A(a1) \to pred_B(b1)) \wedge (pred_A(a2) \to pred_B(b2)) & \text{by de Morgan} \\
= & pred_{A\Rightarrow B}(a1 \rightharpoonup b1) \wedge pred_{A\Rightarrow B}(a2 \rightharpoonup b2) & \text{by (14)}
\end{array}
$$

We note $\Cup_{A\Rightarrow B}$ the last useful operation that we can form using the union and intersection operations of A and B.

$$(a1 \to b1) \Cup_{A\Rightarrow B} (a2 \to b2) \triangleq (a1 \sqcup_A a2) \to (b1 \sqcup_B b2) \tag{26}$$

This operation performs a union on both sides of the implication, we will see shortly why we need it.

Denote I_1 (resp. I_2) the implication lattice $A \Rightarrow B$ (resp. $C \Rightarrow D$). We call implication product of I_1 and I_2, and denote $I_1 \otimes I_2$, a new implication lattice built from I_1 and I_2. Ordering is pair-ordering, where a pair is greater than another one if it is the case for both their left and right elements. Union and intersection are defined pairwise, like least and greatest elements, as in a cartesian or reduced product:

$$
\begin{aligned}
&(a1 \rightharpoonup b1, c1 \rightharpoonup d1) \sqcup_{I_1\otimes I_2} (a2 \rightharpoonup b2, c2 \rightharpoonup d2) \\
&\qquad = ((a1 \rightharpoonup b1) \sqcup_{A\Rightarrow B} (a2 \rightharpoonup b2), (c1 \rightharpoonup d1) \sqcup_{C\Rightarrow D} (c2 \rightharpoonup d2))
\end{aligned} \tag{27}
$$

$$
\begin{aligned}
&(a1 \rightharpoonup b1, c1 \rightharpoonup d1) \sqcap_{I_1\otimes I_2} (a2 \rightharpoonup b2, c2 \rightharpoonup d2) \\
&\qquad = ((a1 \rightharpoonup b1) \sqcap_{A\Rightarrow B} (a2 \rightharpoonup b2), (c1 \rightharpoonup d1) \sqcap_{C\Rightarrow D} (c2 \rightharpoonup d2))
\end{aligned} \tag{28}
$$

This accounts for an easy modularization and parameterization of the analysis.

Theorem 3 $(I_1 \otimes I_2, \leq_{I_1\otimes I_2}, \bot_{I_1\otimes I_2}, \top_{I_1\otimes I_2}, \sqcup_{I_1\otimes I_2}, \sqcap_{I_1\otimes I_2})$ *forms a complete lattice.*

Proof. Same as cartesian or reduced product. □

We define concretization and abstraction over this product as follows:

$$pred_{I_1\otimes I_2}(a \rightharpoonup b, c \rightharpoonup d) \triangleq (pred_A(a) \wedge pred_C(c)) \to (pred_B(b) \vee pred_D(d)), \tag{29}$$

$$aval_{I_1\otimes I_2}(f) \triangleq (aval_{I_1}(f), aval_{I_2}(f)). \tag{30}$$

Theorem 4 *$I_1 \otimes I_2$ with concretization $pred_{I_1 \otimes I_2}$ defines a valid abstraction of Φ.*

Proof. First, we prove that $pred_{I_1 \otimes I_2} \circ aval_{I_1 \otimes I_2}$ over-approximates identity. Here, the canonical form $\psi \rightarrow \phi$ of a formula f must respect equation 21 for both lattices A and C. Again, $True \rightarrow f$ is a valid such formula.

$$
\begin{array}{lll}
 & pred_{I_1 \otimes I_2} \circ aval_{I_1 \otimes I_2}(\psi \rightarrow \phi) & \\
= & pred_{I_1 \otimes I_2}(aval_{I_1}(\psi \rightarrow \phi), aval_{I_2}(\psi \rightarrow \phi)) & \text{by (30)} \\
= & pred_{I_1 \otimes I_2}(aval_A(\psi) \rightharpoonup aval_B(\phi), aval_C(\psi) \rightharpoonup aval_D(\phi)) & \text{by (23)} \\
= & (pred_A \circ aval_A(\psi) \wedge pred_C \circ aval_C(\psi)) \rightarrow (pred_B \circ aval_B(\phi) \vee pred_D \circ aval_D(\phi)) & \\
 & & \text{by (29)} \\
\Leftarrow & (pred_A \circ aval_A(\psi) \wedge pred_C \circ aval_C(\psi)) \rightarrow \phi & \text{by (4)} \\
= & \psi \rightarrow \phi & \text{by (21)}
\end{array}
$$

We first need to prove that $\sqcup_{I_1 \otimes I_2}$ over-approximates the concrete union.

$$
\begin{array}{lll}
 & pred_{I_1 \otimes I_2}((a1 \rightharpoonup b1, c1 \rightharpoonup d1) \sqcup_{I_1 \otimes I_2} (a2 \rightharpoonup b2, c2 \rightharpoonup d2)) & \\
= & pred_{I_1 \otimes I_2}((a1 \rightharpoonup b1) \sqcup_{A \Rightarrow B} (a2 \rightharpoonup b2), (c1 \rightharpoonup d1) \sqcup_{C \Rightarrow D} (c2 \rightharpoonup d2)) & \\
 & & \text{by (27)} \\
= & pred_{I_1 \otimes I_2}((a1 \sqcap_A a2) \rightharpoonup (b1 \sqcup_B b2), (c1 \sqcap_C c2) \rightharpoonup (d1 \sqcup_D d2)) & \text{by(16)} \\
= & pred_A(a1 \sqcap_A a2) \wedge pred_C(c1 \sqcap_C c2) \rightarrow pred_B(b1 \sqcup_B b2) \vee pred_D(d1 \sqcup_D d2) & \\
 & & \text{by (29)} \\
\Leftarrow & (pred_A(a1) \wedge pred_A(a2) \wedge pred_C(c1) \wedge pred_C(c2)) \rightarrow (pred_B(b1) \vee pred_B(b2) \vee & \\
 & pred_D(d1) \vee pred_D(d2)) & \text{by (2) and (3)} \\
= & ((pred_A(a1) \wedge (pred_C(c1)) \rightarrow (pred_B(b1) \vee pred_D(d1)) \vee (pred_A(a2) \wedge pred_C(c2)) \rightarrow & \\
 & (pred_B(b2) \vee pred_D(d2)) & \text{by de Morgan} \\
= & pred_{I_1 \otimes I_2}(a1 \rightharpoonup b1, c1 \rightharpoonup d1) \vee pred_{I_1 \otimes I_2}(a2 \rightharpoonup b2, c2 \rightharpoonup d2) & \text{by (29)}
\end{array}
$$

We also prove that $\sqcap_{I_1 \otimes I_2}$ under-approximates the concrete intersection.

$$
\begin{array}{lll}
 & pred_{I_1 \otimes I_2}((a1 \rightharpoonup b1, c1 \rightharpoonup d1) \sqcap_{I_1 \otimes I_2} (a2 \rightharpoonup b2, c2 \rightharpoonup d2)) & \\
= & pred_{I_1 \otimes I_2}((a1 \rightharpoonup b1) \sqcap_{A \Rightarrow B} (a2 \rightharpoonup b2), (c1 \rightharpoonup d1) \sqcap_{C \Rightarrow D} (c2 \rightharpoonup d2)) & \\
 & & \text{by (28)} \\
= & pred_{I_1 \otimes I_2}((a1 \sqcup_A a2) \rightharpoonup (b1 \sqcap_B b2), (c1 \sqcup_C c2) \rightharpoonup (d1 \sqcap_D d2)) & \text{by(17)} \\
= & pred_A(a1 \sqcup_A a2) \wedge pred_C(c1 \sqcup_C c2) \rightarrow pred_B(b1 \sqcap_B b2) \vee pred_D(d1 \sqcap_D d2) & \\
 & & \text{by(29)} \\
\Rightarrow & ((pred_A(a1) \vee pred_A(a2)) \wedge (pred_C(c1) \vee pred_C(c2))) \rightarrow ((pred_B(b1) \wedge pred_B(b2)) \vee & \\
 & (pred_D(d1) \wedge pred_D(d2))) & \text{by (2) and (3)} \\
= & ((\overline{pred_A(a1)} \wedge \overline{pred_A(a2)}) \vee (\overline{pred_C(c1)} \wedge \overline{pred_C(c2)})) \vee ((pred_B(b1) \wedge pred_B(b2)) \vee & \\
 & (pred_D(d1) \wedge pred_D(d2))) & \text{by de Morgan} \\
\Rightarrow & (\overline{pred_A(a1)} \vee \overline{pred_C(c1)} \vee pred_B(b1) \vee pred_D(d1)) \wedge (\overline{pred_A(a2)} \vee \overline{pred_C(c2)} \vee & \\
 & pred_B(b2) \vee pred_D(d2)) & \text{by de Morgan} \\
= & ((pred_A(a1) \wedge (pred_C(c1)) \rightarrow (pred_B(b1) \vee pred_D(d1)) \wedge (pred_A(a2) \wedge pred_C(c2)) \rightarrow & \\
 & (pred_B(b2) \vee pred_D(d2)) & \text{by de Morgan} \\
= & pred_{I_1 \otimes I_2}(a1 \rightharpoonup b1, c1 \rightharpoonup d1) \wedge pred_{I_1 \otimes I_2}(a2 \rightharpoonup b2, c2 \rightharpoonup d2) & \text{by (29)}
\end{array}
$$

□

From now, $\mathcal{L}$ stands for our background abstract lattice; we assume it is stable by intersection. It is intended to replace A and B in the implication lattice above.

3.5 Inferring loop invariants and function preconditions

It is common to use forward AI to discover program invariants, as done in [3] or [22]. It consists in propagating through the control-flow graph an over-approximation of the sets of possible data values at each point until it converges. This is not sufficient in our modular setting, since a forward analysis cannot generate function preconditions that guarantee correctness. Our need for a contextual analysis makes things even more complex: plain AI works over convex sets; complex techniques like disjunctive completion or dynamic value trace partitioning are needed in order to make AI contextual. In order to build the desired modular and contextual analysis, we use AI in a novel way described below.

In the following, we identify an abstract value produced by AI and the logical formula describing this abstract value. This allows us to switch back-and-forth between the AI view and the predicate calculus view. At each program point a memory access is performed, we can express its safety as the validy of the formula ϕ given in Figures 21 and 22. Whenever the invariant I computed by AI logically implies the formula ϕ created this way, the memory access is safe.

We used in our implementation the domain of octagons [28]. This minimal relational domain is able to express all constraints of the form $\pm x \pm y \leq c$ where x and y are variables and c is an integer constant. This seems to be the most interesting domain for memory analysis, as argued in [22], although we might need a full relational domain when considering programs with casts and unions. It is sufficient for exactly representing the linear conditions that we have introduced so far, either for our memory model, for transfer functions over this model and for expressing the safety of accesses, when the index i is a variable or an integer constant. Our method applies equally to any domain, with some of the most interesting features (like loop elimination) applying only to relational domains.

3.5.1 Initial forward propagation

We use forward AI as a first step to generate invariants at each program point. During this propagation, we take into account both *assumptions*, whether the precondition of the function analyzed or the postconditions of called functions, and *assertions*, either calls to the C function `assert` or logical assertions. This propagation step uses widening on loop backedges as a means to converge. Formulas expressing the safety of memory accesses are used here as logical assertions, so that a same memory access is not checked twice. On our example, initial forward propagation produces the annotated code shown in Figure 24.

Further forward propagations build on the invariants computed by initial forward propagation. We use this peculiarity to define a one-pass refinement operator, i.e. an operator which replaces the normal looping and widening process, so that iterating around loops is not needed anymore. The simplest one-pass refinement operator consists in forgetting at

```
//@ requires: full_separated(dest, src)
char* foo(char* dest, char* src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   //@ invariant: 0 ≤ cur_offset = src_self_offset ≤ −s_self_offset
   while (s + s_self_offset−− > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 24: Example (cont.): after initial forward propagation

loop entry any new information on the variables modified in the loop. This is not sufficient to propagate new information on variables modified in the loop. For these variables, a better one-pass refinement operator consists in performing a union between the new abstract state at loop entry from the one-pass forward propagation, and the abstract state at loop end from the initial normal forward propagation. In our context of use, new information forward propagated from a newly computed precondition or loop invariant is usually known to be true at loop end after initial forward propagation. This is because we use memory safety conditions as logical assertion during initial forward propagation. It makes our one-pass refinement operator very effective at discovering new loop invariants.

3.5.2 Starting point: memory safety conditions

At label L where a memory access is performed, we consider the invariant I computed by initial forward propagation and the safety condition ϕ. At this point, it is sufficient to prove validity of implication $I \rightarrow \phi$ to prove the access safe.

If ϕ can be represented exactly in $\mathcal{L}$, proving $I \rightarrow \phi$ amounts to an inclusion query between abstract values. Otherwise, we consider the disjuncts $\overline{\phi_i}$ in $\overline{\phi}$ and perform emptiness queries on the conjuncts $\overline{\phi_i} \wedge I$. In our case, this would be probably more efficient to resort to a simplex algorithm to solve these queries, as the octagon domain generates only linear constraints.

3.5.3 Backward one-pass propagation

If the implication formula $I \rightarrow \phi$ cannot be proved valid at label L where a memory access is performed, we form the corresponding abstract value in the implication lattice $\mathcal{L} \Rightarrow \mathcal{L}$, and try to propagate this value up in the code, in order either to prove it or to generate loop invariants and preconditions sufficient to prove it. This is not sound, as the implication

$I \rightarrow \phi$ is not guaranteed to be a valid formula: even if the memory access is safe, it can be due to some program logic that AI could not capture in I. As a consequence, the formulas $\psi \rightarrow \phi$ obtained by backward propagation of $I \rightarrow \phi$ are not guaranteed either to be valid formulas. Inferring loop invariants and preconditions from these formulas is only a heuristic method, with no completeness or soundness guarantee. Despite that, we will see it is very efficient at guessing appropriate loop invariants and preconditions for the kind of programs we consider.

Backward propagation in AI has already been used for various reasons. Miné mentions it in his description of the octagon domain [28] as a means to refine the invariants computed by forward AI. Rival [33] uses backward AI to understand the origin of the alarms generated by forward AI. The closest use to ours is the work of Bourdoncle who tries in [7] to propagate backward an over-approximation of the states that do not lead to an error. Although his work inspired ours, our back-and-forth propagation is completely different from his backward propagation, that is simply the backward version of normal forward propagation. We will explain in Section 6 why our method supersedes his. Our goal is to propagate simultaneously:

1. an over-approximation of the set of states from which it is possible to reach label L,
2. and an over-approximation of the set of states from which all sequences of steps that reach label L result in validity of ϕ.

At label L, the first set of states corresponds to I and the second set of states corresponds to ϕ. We see immediately that the desired propagation corresponds to operation $\Cup_{A \Rightarrow B}$ on the implication lattice $\mathcal{L} \Rightarrow \mathcal{L}$. We delay to Section 4 the explainations on how we implement this efficiently with octagons.

Except for loops, nearly classical backward transfer functions are applied on both parts of the implication abstract value. For now, it is sufficient to know that transfer functions on $\psi \rightarrow \phi$ add context to the left-hand part ψ only, while modifying the right-hand part ϕ to get the correct over-approximation. This adds path-sensitive information to the formula, so that it is easier to prove it valid. In particular, a contextual precondition for the function is inferred if the formula $I \rightarrow \phi$ propagates all the way up to the function entry, while still being in the form of an implication $\psi \rightarrow \phi$, with ψ neither empty not implied by an existing precondition. The formula $\psi \rightarrow \phi$ becomes a new precondition for the function.

3.5.4 Loop elimination

Two cases are possible once the abstract value representing an implication formula $\psi \rightarrow \phi$ reaches a loop head during backward propagation.

1. The right-hand part ϕ of the implication does not mention variables modified in the loop. It is sufficient to forget those variables in the left-hand part ψ of the implication to produce a candidate loop invariant $\psi' \rightarrow \phi$.
2. The right-hand part ϕ of the implication mentions variables modified in the loop. In most cases, simply forgetting those variables in the formula removes the right-hand part altogether. We need smarter elimination strategies.

We define several elimination strategies or heuristics to deal with case 2. Suppose there is only one variable v that is both modified in the loop and present in ϕ. Dealing with more than one such variable consists in applying the same treatment for each one. We now detail each heuristic with some explainations for why it might be a good guess at the correct invariant.

1. Fourier-Motzkin elimination
   We use it in the cases where v has a lower bound (resp. an upper bound) in both parts of the implication. These bound can easily be made non-strict by incrementing or decrementing them by one, since the underlying type is integer. We rewrite the implication formula by ignoring all other sub-formulas:

   $$v \geq lhs_bound \rightarrow v \geq rhs_bound \tag{31}$$

   Validity of equation 31 where v is universally quantified is equivalent to the following formula, obtained through Fourier-Motzkin elimination of v.

   $$lhs_bound \geq rhs_bound \tag{32}$$

2. Transitive elimination
   We use it in the cases where v has a lower bound (resp. an upper bound) in the left-hand part of the implication, and an upper bound (resp. a lower bound) in the right-hand part of the implication. Again, we rewrite the implication formula by ignoring all other sub-formulas:

   $$v \geq lhs_bound \rightarrow v \leq rhs_bound \tag{33}$$

   If the context for this implication is reachable (there exists v such that $v \geq lhs_bound$), then for this particular value of v we get an equivalent formula:

   $$v \geq lhs_bound \wedge v \leq rhs_bound \tag{34}$$

   Validity of equation 34 where v is existentially quantified is equivalent to the following formula, obtained through Fourier-Motzkin elimination of v.

   $$lhs_bound \leq rhs_bound \tag{35}$$

3. Min-max Fourier-Motzkin elimination
   We use it when more than one lower bound (resp. more than one upper bound) is available for Fourier-Motzkin elimination, as described above. E.g., the implication formula rewritten by ignoring all other sub-formulas could look like:

   $$v \geq lhs_bound1 \wedge v \geq lhs_bound2 \rightarrow v \geq rhs_bound \tag{36}$$

In that case, we recover the normal Fourier-Motzkin elimination by using the special meta-variable generators *min* and *max* described in Section 3.3:

$$v \geq max(lhs_bound1, lhs_bound2) \rightarrow v \geq rhs_bound \tag{37}$$

Applying the elimination strategy seen above, we get:

$$max(lhs_bound1, lhs_bound2) \geq rhs_bound \tag{38}$$

4. Min-max Transitive elimination
   We use it when more than one lower bound (resp. more than one upper bound) is available for Transitive elimination, as described above. E.g., the implication formula rewritten by ignoring all other sub-formulas could look like:

$$v \geq lhs_bound1 \wedge v \geq lhs_bound2 \rightarrow v \leq rhs_bound \tag{39}$$

   In that case, we recover the normal Transitive elimination by using the special meta-variable generators *min* and *max* described in Section 3.3:

$$v \geq max(lhs_bound1, lhs_bound2) \rightarrow v \leq rhs_bound \tag{40}$$

   Applying the elimination strategy seen above, we get:

$$max(lhs_bound1, lhs_bound2) \leq rhs_bound \tag{41}$$

In our experiments, we noticed that applying Fourier-Motzkin (normal or min-max version) whenever possible gives better results. Therefore we favor this heuristic over Transitive elimination. The presence of variables $arrlen(p)$ in the left-hand part of the implication caused loop invariants of the form $arrlen(p) < s \rightarrow \phi$ to be inferred. These formulas are totally meaningless for a C program, because the programmer has no way to test the allocated size of a memory block after it has been allocated. Therefore we remove all variables $arrlen(p)$ from the left-hand part of the implication before elimination, to increase the odds of finding a meaningful loop invariant. Overall, applying any of these eliminations allows us to avoid iterating around loops, which is a key feature for scalability.

3.5.5 Contextual one-pass forward propagation

Once loop invariants and preconditions have been generated by backward propagation, they can be considered for a forward one-pass propagation through the code. In the context of AI×DV, the goal might be to prove as many memory accesses as possible. In both verification frameworks it avoids performing backward propagation again from a memory access that could be proved safe using the newly generated logical annotations. This leads to a back-and-forth algorithm where we consider each memory access in turn.

Forward propagation of implication abstract values $a \rightharpoonup b$ uses the union and intersection operations of the implication lattice as defined above. Contrary to what is done during

backward propagation, context is not added to the left-hand side a of the implication. Indeed, the implication abstract value is kept as simple as possible, while computing an under-approximation of its left-hand side and an over-approximation of its right-hand side. This minimizes the possible loss of information during unions where paths merge. Aside from this implication value, we still consider the normal abstract value without implication used e.g., during initial forward propagation. Whenever the left-hand side $pred(a)$ of the implication is implied by the normal abstract value at this point, we add the corresponding right-hand side b of the implication to the normal abstract value. The declared objective of this step is to add enough information to the abstract value describing the current state so that the new computed invariant I' implies the safety of the memory access that started this back-and-forth pass (and maybe others too), or equivalently that $pred(I') \rightarrow \phi$ is valid.

We can now express the refinement operator for implication abstract values. Let I and J be the normal abstract values at loop entry and loop end, $a \rightharpoonup b$ the implication abstract value reaching loop entry. Our modified one-pass refinement operator returns as new invariant for the loop the value:

$$R_{\mathcal{L}\Rightarrow\mathcal{L}}(I, J, a \rightharpoonup b) \triangleq a \rightharpoonup aval_{\mathcal{L}}(pred_{\mathcal{L}}((I \sqcap a \sqcap b) \sqcup J) \setminus pred_{\mathcal{L}}(I)), \tag{42}$$

where $\setminus$ is a logical subtraction, such that $\phi\setminus\psi$ removes from the conjuncts of ϕ those conjuncts also in ψ. Remember that $\mathcal{L}$ is stable by intersection.

Theorem 5 *One-pass refinement operators $R_{\mathcal{L}\Rightarrow\mathcal{L}}$ correctly over-approximates the most precise loop invariant.*

Proof.

$$\begin{array}{lll}
 & pred_{\mathcal{L}\Rightarrow\mathcal{L}}(R_{\mathcal{L}\Rightarrow\mathcal{L}}(I, J, a \rightharpoonup b)) & \\
= & pred_{\mathcal{L}\Rightarrow\mathcal{L}}(a \rightharpoonup aval_{\mathcal{L}}(pred_{\mathcal{L}}((I \sqcap a \sqcap b) \sqcup J) \setminus pred_{\mathcal{L}}(I))) & \text{by (42)} \\
\leftarrow & pred_{\mathcal{L}}(a) \rightarrow pred_{\mathcal{L}} \circ aval_{\mathcal{L}}(pred_{\mathcal{L}}((I \sqcap a \sqcap b) \sqcup J) \setminus pred_{\mathcal{L}}(I)) & \text{by (15)} \\
\leftarrow & pred_{\mathcal{L}}(a) \rightarrow (pred_{\mathcal{L}}((I \sqcap a \sqcap b) \sqcup J) \setminus pred_{\mathcal{L}}(I)) & \text{by (4)} \\
\leftarrow & pred_{\mathcal{L}}(a) \rightarrow pred_{\mathcal{L}}((I \sqcap a \sqcap b) \sqcup J) &
\end{array}$$

At loop entry, current invariant is:

$$Inv_{entry} = pred_{\mathcal{L}}(I) \wedge pred_{\mathcal{L}\Rightarrow\mathcal{L}}(a \rightharpoonup b) = pred_{\mathcal{L}}(I) \wedge (pred_{\mathcal{L}}(a) \rightarrow pred_{\mathcal{L}}(b)) \tag{43}$$

Therefore, at loop entry:

$$\begin{array}{lll}
 & pred_{\mathcal{L}\Rightarrow\mathcal{L}}(R_{\mathcal{L}\Rightarrow\mathcal{L}}(I, J, a \rightharpoonup b)) & \\
\leftarrow & pred_{\mathcal{L}}(a) \rightarrow pred_{\mathcal{L}}(I \sqcap a \sqcap b) & \text{by (2)} \\
= & pred_{\mathcal{L}}(a) \rightarrow (pred_{\mathcal{L}}(I) \wedge pred_{\mathcal{L}}(a) \wedge pred_{\mathcal{L}}(b)) & \text{by (5)} \\
\leftarrow & pred_{\mathcal{L}}(I) \wedge (pred_{\mathcal{L}}(a) \rightarrow pred_{\mathcal{L}}(b)) & \\
= & Inv_{entry} & \text{by (43)}
\end{array}$$

At loop end, current invariant is:

$$Inv_{end} = pred_{\mathcal{L}}(J) \tag{44}$$

Therefore, at loop end:

$$
\begin{array}{lll}
 & pred_{\mathcal{L}\Rightarrow\mathcal{L}}(R_{\mathcal{L}\Rightarrow\mathcal{L}}(I, J, a \rightharpoonup b)) & \\
\leftarrow & pred_{\mathcal{L}}(a) \rightarrow pred_{\mathcal{L}}(J) & \text{by (2)} \\
\leftarrow & pred_{\mathcal{L}}(J) & \\
= & Inv_{end} & \text{by (44)}
\end{array}
$$

□

It can be seen as the counterpart for implication abstract values of the one-pass refinement seen in Section 3.5.1. It has the same good property that allows discovering new loop invariants: new forward propagated information that is already known to be true at loop end is kept in the loop invariant.

3.5.6 Flashback: the case of strings

As seen in equation 11, being a string is equivalent to some linear inequation involving the length of the string. This inequation can be represented in the abstract domain we work with, using $strlen(p)$ as a meta-variable, as seen in Section 3.2. Therefore, we can use the back-and-forth propagation just described to infer loop invariants and preconditions that are likely to guarantee a pointer is a string at some point in the program. On our example, string inference using backward propagation produces the annotated code shown in Figure 25.

```
//@ requires: full_separated(dest, src)
//@           ∧ 1 ≤ s → 0 ≤ strlen(src)
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   //@ invariant: 0 ≤ cur_offset = src_self_offset ≤ −s_self_offset
   //@            ∧ 1 ≤ s → 0 ≤ strlen(src)
   while (s + s_self_offset−− > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 25: Example (cont.): after string inference (backward only)

Before propagating forward these preconditions and loop invariants, we should look more closely at what it means to be a string in C. Using idiom 5, we can add information on $strlen(s)$ whenever testing the (non-)nullity of some string access $s[i]$. Indeed, the nullity of $s[i]$ can be understood as the equality $i = strlen(s)$ and the non-nullity of $s[i]$ as the inequality $i < strlen(s)$. This must be added as an assumption on each branch that originates

in a (non-)nullity test of $s[i]$. It is not an assertion, that could not be used in computing loop invariants, as seen when discussing initial forward propagation.

On our example, the initial forward propagation produced $i \leq strlen(src)$ at loop end, and the current contextual forward propagation gives invariant $1 \leq s \rightarrow i \leq strlen(src)$ at loop entry. If this invariant was not contextual (e.g., for C function *strlen*, we get simply $i \leq strlen(src)$), using the improved one-pass refinement operator that we saw for initial forward propagation would produce a generalized loop invariant $i \leq strlen(src)$. With an additionnal contextual part, this does not work. We built a one-pass generalizing refinement operator that does the same for contextual formulas like here, in most simple cases. On our example, using this one-pass generalizing refinement operator, the contextual forward propagation that follows string inference produces the annotated code shown in Figure 26. This is sufficient to prove the correctness of accesses to *src*, by AI only.

```
//@ requires: full_separated(dest, src)
//@           ∧ 1 ≤ s → 0 ≤ strlen(src)
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   //@ invariant: 0 ≤ cur_offset = src_self_offset ≤ −s_self_offset
   //@            ∧ 1 ≤ s → −s_self_offset ≤ strlen(src)
   while (s + s_self_offset−− > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 26: Example (cont.): after string inference

3.6 Proving memory safety

On our example, applying our back-and-forth methods produces the annotated code shown in Figure 27. If we consider the AI×DV verification framework, AI alone is able to prove that function *foo* is memory safe with these annotations. If we consider the AI+DV verification framework, Simplify and Yices prove all safety verification conditions generated using these annotations.

```
//@ requires: full_separated(dest, src)
//@          ∧ 1 ≤ s → 0 ≤ strlen(src)
//@          ∧ 1 ≤ s → min(s, strlen(src)) < arrlen(dest)
//@          ∧ s ≤ 0 → 0 < arrlen(dest)
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   //@ invariant: 0 ≤ cur_offset = src_self_offset ≤ −s_self_offset
   //@          ∧ 1 ≤ s → −s_self_offset ≤ strlen(src)
   //@          ∧ 1 ≤ s → min(s, strlen(src)) < arrlen(dest)
   //@          ∧ s ≤ 0 → 0 < arrlen(dest)
   while (s + s_self_offset−− > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 27: Example (cont.): after back-and-forth inference

3.7 Adding useful predicates to context

The preconditions generated so far are not completely satisfactory. The nullity test on *dest* is not taken into account, as it cannot be represented in our abstract domain of octagons. Treating *dest* as an integer variable would allow us to represent condition $dest = 0$, but not condition $dest \neq 0$ that we are interested in.

To include arbitrary equalities and disequalities in our context, we add a very basic predicate domain that only follows equalities and disequalities. We form a reduced product between this domain and the octagon domain used for the left-hand part of an implication. On our example, using this predicate domain produces the desired preconditions, as shown in Figure 28.

3.8 Overall method

Finally, our method consists in the three propagation phases that are sketched in bold arrows on the diagrams in Figure 29, where the last two phases are called in turn for each memory access. We detail the precise algorithm in Figure 30.

```
//@ requires: full_separated(dest, src)
//@          ∧ (dest ≠ 0 ∧ 1 ≤ s) → 0 ≤ strlen(src)
//@          ∧ (dest ≠ 0 ∧ 1 ≤ s) → min(s, strlen(src)) < arrlen(dest)
//@          ∧ (dest ≠ 0 ∧ s ≤ 0) → 0 < arrlen(dest)
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   int src_self_offset = 0;
   int s_self_offset = 0;
   int cur_offset = 0;
   if (dest == 0) return 0;
   //@ invariant: 0 ≤ cur_offset = src_self_offset ≤ −s_self_offset
   //@          ∧ 1 ≤ s → −s_self_offset ≤ strlen(src)
   //@          ∧ 1 ≤ s → min(s, strlen(src)) < arrlen(dest)
   //@          ∧ s ≤ 0 → 0 < arrlen(dest)
   while (s + s_self_offset−− > 0 && src[src_self_offset]) {
      dest[cur_offset++] = src[src_self_offset++];
   }
   dest[cur_offset] = '\0';
   return dest;
}
```

Figure 28: Example (cont.): with basic predicate domain

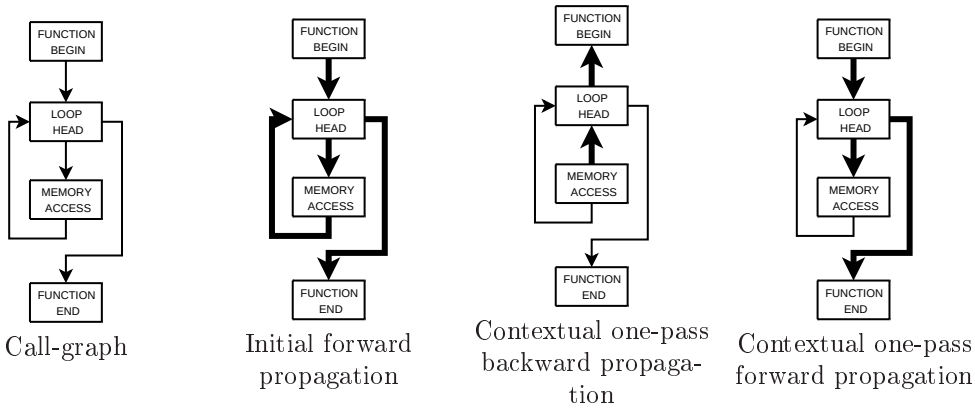

Figure 29: Sketch of the method

4 Implementation

Our implementation is roughly 10.000 lines of OCAML for the plugin part inside Caduceus, and a few hundreds lines of C to patch the available octagon library [28]. Both modules communicate using the OCAML binding of the octagon library. Our plugin and octagon

1. perform local aliasing analysis and transform program
2. perform interprocedural separation analysis and annotate program
3. perform initial forward propagation and add loop invariants
4. define procedure **back-and-forth** (formula ϕ, program point L) as
   (a) call I the invariant computed by AI at L
   (b) build the formula $I \rightarrow \phi$
   (c) propagate $I \rightarrow \phi$ backward from L in a single pass:
   use backward AI except for loops
   use special elimination for loops
   (d) **if** formula propagated still contextual at function or loop beginning **then**
   i. add it as new function precondition or new loop invariant
   ii. propagate these new assumptions forward using AI
   iii. use the implication domain transfer functions and operations
5. **for** each string typestate (idioms (2) and (3)) in the program **do**
   if string typestate proved by AI
   then do nothing
   else call **back-and-forth** with ϕ whose validity implies string typestate
   done
6. **for** each memory access in the program **do**
   if memory access proved by AI
   then do nothing
   else call **back-and-forth** with ϕ whose validity implies safety for the access
   done
7. **if** doing AI×DV
   then set as definitely proved the memory accesses proved by AI
8. **if** some memory accesses and annotations not proved
   then call DV on the generated verification conditions

Figure 30: Method for proving the memory safety of C pointer programs

patch, as well as the Caduceus and WHY tools, are freely available from [27]. We support the full ANSI C language accepted by Caduceus (that excludes casts and unions) by limiting our analysis to functions that contain statements of the kind described in this paper. The dynamic allocation we described using operator *new* is in fact recognized as calls to standard C library function *malloc* of the form:

$$new\ t[n] \equiv (t*)\ malloc(sizeof(t)\ *\ n)$$

All basic C types are supported, as well as arrays. Loops may be of any form (*for*, *while*, $do - while$) and contain *break*, *continue* or *goto* statements that do not enter scopes.

The implication domain is based on the octagon one. Instead of defining the left part of the implication to be an octagon and the right part another octagon, we pack both parts in one octagon. This is an important decision both for scalability and simplicity. The drawback of this decision is that we cannot represent formulas like $x > 0 \rightarrow x > 10$, because the same inequality is used on both sides of the formula. This may seem like an unimportant matter, but it prevented us from representing nullity of pointers as the nullity of their allocated size, because then we generated formulas like the above, with $arrlen(p)$ for x. This choice respects relations 15 and 18.

In order to distinguish between inequalities that belong to the left part and inequalities that belong to the right part, we simply tag inequalities from the right part (usually one or two only). One of the most important operation on octagons, the closure computation, has to be restricted to the untagged inequalities. Closing an octagon derives the tighest possible bounds on each considered inequality of the form $\pm x \pm y \leq c$. Our restriction prevents merging inequalities from the left and the right parts of an implication. We use in our implementation the implication product of this octagon implication lattice I_{oct} and the lattice $P_{=\neq}$ of simple predicates based on equalities and disequalities. This is possible because the latter is isomorphic to the constant implication lattice $P_{=\neq} \Rightarrow False$.

The transfer function for test only adds untagged inequalities, which ensures that we only add context to the left part of an implication, as said in Section 3.5. The transfer function for assignment modifies all existing inequalities, but adds only untagged inequalities. Modified inequalities keep their tagging status. This again ensures we keep the right part of the implication minimal while still being correct.

Each implication formula set as function precondition or loop invariant by our method receives a new identifying number, so that forward propagation of more than one implication abstract value correctly unions corresponding abstract values after branching.

Conversion between octagons and first-order formulas is quite simple. Starting from a first-order formula, we consider each one of its conjuncts and constrain an initially full octagon ($\top$) with these conjuncts, using the transfer function for test. Starting from an octagon, we convert it to hollow form [28], a kind of minimization, and we output the conjunction of the inequalities in this minimal form. This minimization step is crucial to make elimination work properly.

In this paper, we only used loop invariants. In our implementation, we distinguish between loop (asserted) invariants and loop assumed invariants. The *assumed* version is

used to store invariants proved by AI, so that we do not need to prove them again using DV in the context of AI×DV. This makes it possible to fully prove functions with loops using only AI. The normal loop invariant then stores the invariant part that could not be proved by AI, either because it was provided by the programmer, or because it was inferred by our method but not proved by forward propagation.

5 Experiments

The standard string library as defined by Ansi C presents a good mix of pointer and string manipulations, with many implicit preconditions only given in textual description, like the overlapping conditions. Since available implementations are heavily optimized, using bit-field manipulations or assembly code, we hand-coded a forward implementation of header string, that would be both simple and idiomatic. Since Caduceus does not accept casts, we modified the signature of string functions involving $void*$ parameters or returns, to replace them by $char*$. To infer the subtle invariants used by these functions, we had to take into account the laziness of boolean operations in C. Indeed, it is quite common in C to write tests where the right operand of a boolean operation is safe only if its left operand has an expected outcome (true or false).

On this implementation, we successfully generated the necessary annotations and automatically proved memory safety for 18 functions out of a total of 22. The four remaining functions are `strcat` and `strncat`, which require inferring linear inequations involving three variables (which is not possible with the octagon domain only) and `strtok` and `strerr`, which require inferring global invariants. With additional hand-written annotations, we also verified automatically these four functions.

As an example of an apparently complex yet correct precondition generated by our method, here is the precondition we generate for the function `strncpy` (which uses logical function *min* presented in Section 3.3):

//@ **requires** $(1 \leq n \rightarrow 0 \leq strlen(src))$
//@ $\wedge\ (1 \leq n \wedge 0 \leq strlen(src)) \rightarrow min(n-1, strlen(src)) < arrlen(dest)$
//@ $\wedge\ (2 \leq n \rightarrow n \leq arrlen(dest))$
//@ $\wedge\ full_separated(dest, src)$

A quick case analysis on the value of n leads to the equivalent simpler formula, which looks more like a valid precondition for `strncpy` that a programmer would specify:

//@ **requires** $(n \leq 0 \vee (0 \leq strlen(src) \wedge n \leq arrlen(dest))) \wedge\ full_separated(dest, src)$

In Figure 31, we compare our results on the standard string library with those of PolySpace C Verifier [31], a popular tool for debugging large pointer-intensive C programs. Although it is mostly used as a debugging tool for standalone programs, PolySpace is really a verification tool based on abstract interpretation. It flags every possible runtime error in the code as an *orange check* while it outputs *green checks* on verified statements. We are only

	Caduceus				PolySpace			
	num mem fun proved	ratio mem fun proved	num str fun proved	ratio str fun proved	num mem fun proved	ratio mem fun proved	num str fun proved	ratio str fun proved
initial	5	100%	13	87%	5	100%	2	13%
rewritten	5	100%	13	87%	5	100%	2	13%

Figure 31: Results on Caduceus and PolySpace

interested here in the color of invalid pointer dereference checks (a.k.a. IDP) which verify the validity of dereferences. This makes it possible to compare the number of functions on which PolySpace only produces green IDP with the number of functions whose verification conditions (VC) are all proved by the automatic provers called by Caduceus.

We first ranked each tool results on our implementation of the standard string library. The comparison was problematic because the modular setting of PolySpace conservatively assumes pointer arguments can be null, which causes many orange IDP. In Caduceus, the generated preconditions contextually rule out those cases. Therefore, we added a `main` function similar to the one generated by PolySpace, except we built a simple valid context for calling functions. In order to distinguish between the benefits we get from our annotation process and those we get from initial local aliasing transformation, we ran this experiment again with initial source code rewritten by our local aliasing transformation.

Comparison is run on self-contained functions not relying on global invariants, which rules out functions `strtok` and `strerror`. Remaining functions are divided into memory or string functions, according to prefix (either "mem" or "str"). PolySpace and Caduceus correctly prove all memory functions. As seen previously, the memory model for strings allows Caduceus to infer correct preconditions for most string functions. On these functions, considered inherently unsafe in some software companies [19], PolySpace conservatively outputs orange IDP. Interestingly, PolySpace does prove function `strncpy`, that only performs bounded accesses on its arguments. PolySpace gives the same results on rewritten code as on initial code, which increases our confidence that our modular and contextual annotation method is responsible for Caduceus good results. Source programs and Caduceus results are available from `http://www.lri.fr/~moy`.

6 Related work

Our work owes much to the early work of Bourdoncle [7]. Long before abstract interpretation was used to completely prove the safety of large programs [5], he attempted to use a combination of forward and backward propagations to locate possible errors in programs, which he called abstract debugging. He too focused on array bound checking. But his backward propagation from assertions (the ones he calls *invariant assertions*) merges the conditions

to reach the program point where the assertion is performed and the conditions to make this assertion valid. These are the parts we separate in our implication abstract value. In his method, it makes it necessary to union the abstract state that we propagate backward with abstract states obtained from forward propagation on other branches, whenever reaching a branching or a loop beginning. This is the reason why he gets his most interesting results with *intermittent assertions*, where the programmer additionnally states that every correct execution through the function should reach that assertion point, and only when forward analysis already determinated accurate invariants. He used the simplest domain for AI, the interval domain. Since then, more accurate and yet efficient domains have been devised for AI, like the one we use, the octagon domain [28].

The next work closest to ours, in its objectives and description, is the modular checker for buffer overflows of [19]. Their annotation language based on *properties*, which would be uninterpreted functions in our setting, allows them to modularly check each function separately, like we do. They devised an unsound inference method to generate automatically some of these properties, like we do. The main difference with our work is that their checker does not verify the safety of memory accesses, but simply reports likely errors. The second important difference is that our method is contextual, theirs is context-sensitive, with their inference method using caller's information to refine callee's one. Our contextual inference allows us to treat functions like *strcpy*, and more importantly to infer different function preconditions for different contexts, both things their method cannot do and that they describe as *unannotatable interfaces*. We must say that the choices they make allow them to apply their checker to millions of lines of legacy code.

Historically, static array bound checking was one of the first difficult properties about programs that people tried to prove, originally on Pascal programs. Cousot and Halbwachs [12] applied abstract interpretation over polyhedrons and managed to check the memory safety of an implementation of heapsort, using manual preconditions. A year later, Suzuki and Ishihata [35] used a computation of weakest preconditions together with deductive verification to check the memory safety of an implementation of tree sort. They already used Fourier-Motzkin elimination at loop beginnings as a refinement heuristic on first-order linear formulas.

Some features of our approach have been used previously for optimization or verification purposes. Gupta [17] studied the forward propagation of assertions to avoid repeated runtime array bound checks. Our work applies the per-path summary approach of debugging tools like PREfix [9] or Archer [37] in a formally justified way amenable to verification. Various verification tools use function symbols much like *arrlen* and *strlen* in order to distinguish between allocated size and string length: Boon [36], Cssv [14], Overlook [2]. All these tools use those function symbols to transform pointer programs into integer ones. BOON then solves the safety problem as an integer constraint problem, CSSV as an integer programming problem and Overlook using AI. A key difference with our approach is the annotation inference method used. BOON only targets violations of string library API, using a coarse unsound flow-insensitive analysis, which makes inference useless. CSSV uses a sound inference method based on weakest preconditions and strongest postconditions that

cannot infer the kind of subtle loop invariants we target. Overlook uses forward AI as its only invariant generation method, which is not sufficient in most cases. Another key difference with us is their coarse treatement of aliasing which is at most a flow-insensitive whole program analysis. This prevents them from automatically verifying programs the way we do.

The necessity for some logical specification of pointer separation in C dates back to C99 standard [21], with the addition of the *restrict* keyword. Various authors have described annotation-based systems to help programmers specify pointer separation [1, 24]. This has been pushed forward as a kind of logic about pointer programs by Reynolds in separation logic [32]. Our local separation analysis was inspired from these works, with the emphasis on locality and automatic inference. In support of our treatment of local aliasing, the frequency of cursor aliasing in real embedded software has been noted by [18].

Many tools based on DV have been used to perform full verification of pointer-intensive programs like the Schorr-Waite algorithm used in garbage collection. In our setting, it was done with Why in [20]. Such full verifications ask for a large amount of user work to provide appropriate annotations, as well as deep understanding of the logical setting of the tool. We only attempt at partial verification of quite simple properties of programs, which is why we expect to need few user interaction. Recently, there have been attempts at using a combination of various proof techniques. In [22], AI is used as a first phase to compute invariants about the program that are used in a second phase to seed the predicate abstraction. Interestingly, they also use the octagon library [28]. In [26], a real feedback loop is built between AI and DV. Starting from a counter-example generated by DV, backward AI is applied, and each loop involved in the counter-example is examined by a special inference method to get more precise loop invariants in the context of the counter-example. Although very promising, this approach suffers from the high cost of calling DV repeatedly. In Boogie, the tool developed in the same team, AI is used only in a first phase to compute invariants about the program [3]. Furthermore, it cannot generate preconditions.

Instead of using DV as safety net when static analysis fails to prove memory accesses safe, a practical approach is to rely on runtime checks. Projects CCured and Cyclone explore this option in two different directions. CCured [30] aims at securing large legacy codes. It is based on a compiler that instruments C code with possibly expensive runtime checks, to ensure memory safety. Although an attempt at proving the correction of accesses in the generated code was done with Blast [4], it was harder than on the original program, due to the instrumentation code added by CCured. Cyclone [23] aims at providing programmers with a safer C language, that still allows efficient code while facilitating porting from C. The restrictions it imposes on programs are interesting points to look at if we want to allow automatic verification of such constructs with our method. Since we too use static analysis in the first phase of our method, our inference method could probably be used with profit in the context of these tools.

7 Future work

In order to deal with real programs, our next steps are the treatment of double indirection, casts and structures. Unions combine the difficulties of casts and structures. Before doing so, we need to gather idioms on the ways C programmers use invariants to prevent memory errors when using double indirection, casts, structures and unions. We will then only lack support for memory deallocation, to be added later on.

A crucial point here is that we need to check the proposed idioms on large bases of programs, if we want them to be reliable. This can only be done through code instrumentation and runtime checking, much as what CCured does. Akin to what is done in CCured, we need first to determine by static analysis which pointers are base pointers and which are strings. This is slightly different from what is done in CCured as CCured only deals with types whereas we deal with typestates. We plan to do this checking in CAT, a tool dedicated to the verification of C programs. Like CCured, it is based on CIL [29].

An interesting feature of our backward propagation that we have not exploited yet is its ability to prove a memory access safe. Indeed, it may be the case that forward AI did not prove an access safe because it merged two paths. When doing backward propagation, it may be the case that on each such path the propagated implication formula becomes true. If no over-approximation was made on the right part of the implication, this is sufficient to prove the memory access safe. This is the case e.g., on the example presented in [26], where the assertion to prove is also extracted from a memory safety condition.

There is also place for improvement in our separation analysis. On our example, it infers $full_separated(dest, src)$, while the most precise annotation would be

$$bound_separated(dest, strlen(src), src, strlen(src)).$$

Using the results of AI would make it possible to bound more precisely the locations read and written by a function, which would make it possible to refine the separation conditions inferred.

Various implementation issues deserve a better treatment. To improve precision, we could use the abstract state computed by forward propagation when performing backward one, as presented in [28]. When efficiency becomes a problem, we should review our implementation of the implication octagon lattice, that currently separates both parts of the implication in OCAML code before calling C functions on each part. We could probably implement all this more efficiently in C.

We mentioned in this paper the degree of confidence that we should have in such an deductive verification, in Section 3.1. Clearly, AI+DV is safer yet less powerful than AI×DV. This allows us to gain confidence in AI×DV by comparing its results with AI+DV when possible. To increase the degree of confidence in AI+DV, we can use different automatic provers on the same verification conditions. The only unsafe part left is the local aliasing transformation, where we modify programs without proving formally that semantics are preserved. It is in fact possible to drop this transformation, all other things being equal, by doing the opposite transformation on the annotations added by our method. On our

```
//@ requires: full_separated(dest, src)
//@          ∧ (dest ≠ 0 ∧ 1 ≤ s) → 0 ≤ strlen(src)
//@          ∧ (dest ≠ 0 ∧ 1 ≤ s) → min(s, strlen(src)) < arrlen(dest)
//@          ∧ (dest ≠ 0 ∧ s ≤ 0) → 0 < arrlen(dest)
char⋆ foo(char⋆ dest, char⋆ src, int s) {
   char⋆ cur = dest;
   if (dest == 0) return 0;
   //@ invariant: 0 ≤ cur − dest = src − old(src) ≤ old(s) − s
   //@          ∧ 1 ≤ s → old(s) − s ≤ strlen(old(src))
   //@          ∧ 1 ≤ s → min(s, strlen(old(src))) < arrlen(dest)
   //@          ∧ s ≤ 0 → 0 < arrlen(dest)
   while (s−− > 0 && ⋆src) {
      ⋆cur++ = ⋆src++;
   }
   ⋆cur = '\0';
   return dest;
}
```

Figure 32: Example (cont.): without program transformation

example, this results in the program shown in Figure 32. The annotation $old(p)$ is used for the value of p at the beginning of the function.

8 Conclusion

We presented a new static method for checking the memory safety of pointer-intensive C programs. Our analysis is incomplete: it expects programs to follow commonly used C idioms. Our analysis is sound: whenever programs do not follow the idioms we selected, it fails to verify them. This method relies on an unsound inference algorithm, based on forward and backward abstract interpretation, to generate the necessary logical annotations, mostly function preconditions and loop invariants. Soundness is obtained by deductive verification. This algorithm was specifically designed for modular and contextual verification. In particular, we crafted a special implication domain for abstract interpretation. We showed how to implement it efficiently using a relational domain, which we did for the octagon domain. This makes the implication domain a cheap disjunctive domain for contextual analysis. We presented two previously unknown lattices for local pointer aliasing and local pointer non-aliasing. We showed that special symbols could be used both as meta-variable generators in abstract interpretation and as uninterpreted functions in deductive verification. This allowed collaboration between abstract interpretation and deductive verification, as well as deductive verification of existential properties (e.g., the fact a pointer is a string). Altogether, we showed our method could be used to prove the memory safety of C pointer programs that

no other available tool can handle. We are looking forward to applying it to more realistic programs, when we have added support for more constructs of C.

References

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proc. PLDI '03*, pages 129–140, New York, NY, USA, 2003.

[2] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static analysis of string manipulations in critical embedded c programs. In *SAS*, pages 35–51, 2006.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. FMCO 2005.

[4] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with blast. In M. Cerioli, editor, *Proc. FASE 2005*, LNCS 3442, pages 2–18, 2005.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI'03*, pages 196–207, San Diego, California, USA, June 7–14 2003.

[6] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[7] François Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *Proc. ESEC '93*, pages 501–516, London, UK, 1993.

[8] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[9] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[10] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.

[11] Sylvain Conchon and Evelyne Contejean. Ergo automatic theorem prover, 2006. `http://ergo.lri.fr/`.

[12] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL '78*, pages 84–96, New York, NY, USA, 1978.

[13] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[14] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *Proc. PLDI '03*, pages 155–167, New York, NY, USA, 2003.

[15] Bruno Dutertre and Leonardo de Moura. *The YICES SMT Solver*. Computer Science Laboratory, SRI International, 2006. `http://yices.csl.sri.com`.

[16] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. pages 15–29.

[17] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proc. PLDI '90*, pages 272–282, New York, NY, USA, 1990.

[18] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *Proc. SIGSOFT '06/FSE-14*, pages 69–80, New York, NY, USA, 2006.

[19] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proc. ICSE '06*, pages 232–241, New York, NY, USA, 2006.

[20] Thierry Hubert and Claude Marché. A case study of c source code verification: the schorr-waite algorithm. In *Proc. SEFM '05*, pages 190–199, Washington, DC, USA, 2005.

[21] International Organization for Standardization (ISO). *The ANSI C standard (C99)*. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`.

[22] Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proc. CAV'06*, volume 4144 of *LNCS*, pages 137–151, 2006.

[23] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proc. 2002 USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002.

[24] David Koes, Mihai Budiu, and Girish Venkataramani. Programmer specified pointer independence. In *Proc. MSP '04*, pages 51–59, New York, NY, USA, 2004.

[25] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *Proc. OOPSLA '00, Minnesota*, pages 105–106, 2000.

[26] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[27] Claude Marché and Jean-Christophe Filliâtre. *The Caduceus tool for the verification of C programs*. `http://why.lri.fr/caduceus/`.

[28] Antoine Miné. The octagon abstract domain. *Higher Order Symb. Comp.*, 19(1):31–100, 2006.

[29] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. CC '02*, pages 213–228, London, UK, 2002.

[30] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proc. POPL '02*, pages 128–139, New York, NY, USA, 2002.

[31] PolySpace Technologies. `http://www.polyspace.com`.

[32] John Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*. Palgrave, 2000.

[33] X. Rival. Understanding the origin of alarms in ASTRÉE. In *12th Static Analysis Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 303–319, London (UK), September 2005.

[34] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[35] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proc. POPL '77*, pages 132–143, New York, NY, USA, 1977.

[36] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS Symposium*, pages 3–17, San Diego, CA, February 2000.

[37] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proc. ESEC/FSE-11*, pages 327–336, New York, NY, USA, 2003.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399