J Comput Virol (2008) 4:5–23
DOI 10.1007/s11416-007-0063-z

SSTIC 2007 BEST ACADEMIC PAPERS

# Static analysis by abstract interpretation: application to the detection of heap overflows

**Xavier Allamigeon · Charles Hymans**

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 3 September 2007 / Published online: 8 November 2007


**Abstract** Several security flaws are the consequence of the presence of programming errors or bugs in software. Heap overflow is the typical example of such errors that allows an attacker to take control of a machine. But considering the growing size and complexity of present software, implementing programs without any error is not an easy task. In this paper, we present a static analysis by abstract interpretation that is focused on security properties: without executing the program, it ensures the absence of any heap overflows.

## 1 Introduction

Since the production of the first integrated circuits, and thanks to the evergoing trend of Moore's law, software have propagated in different technological fields. Nowadays, they play a crucial role in our lives: nuclear installation supervision, fly-by-wire systems in planes, car breaking systems, mobile phones, medical resonance imaging, etc.

Moreover, whatever the application field may be, the size of program codes tends to grow significantly. The evolution of the size of Microsoft operating systems, illustrated in Fig. 1, is representative of this trend. As a more illustrative example, a Boeing 777 aircraft and a recent car, embed 4 and 35 million lines of code respectively [30,40]. While it is most of the time invisible to the eyes, software nonetheless perform increasingly complex tasks and thus the risks caused by software are often under-estimated. Unfortunately, any simple programmation error in an embedded software, may render the product ineffective, or even worst, vulnerable to malicious users.

Even the most vigilant programmers cannot avoid the introduction of programming errors in software containing several millions lines of code. It is estimated that the rate of errors per 100 lines of code, before any verification activity, ranges from 0.5 to 5. The majority of these errors are eliminated by code reviews and testing. But these traditional verification methods lack exhaustivity. Indeed, only 70% of bugs can be discovered by an efficient code review [24]. Moreover, such tests cover only a subset of the possible behaviors of the programs (50% according to [24]). As a result, these verification steps cannot ensure the absence of errors in software, which is not acceptable for critical codes.

*Static analysis* is a method to detect errors in programs, without executing them (only by inspecting their source code). A lot of static analysis tools exist (see Sect. 6). Some of them use *abstract interpretation*, a theory which allows to over-approximate the set of *all* the possible behaviors of a program. The analyses by abstract interpretation are said to be *sound*: they can formally show the absence of errors in a program.[1] Nevertheless, if the performed over-approximations are not precise enough, a *false alarm* (also called *false positive*) may be raised: the analysis points out errors that do not actually exist at runtime. Consequently, the challenge is to design fully automatic analyses with an optimal tradeoff between precision and efficiency to handle programs of several millions of lines of code.

This work have been partly supported by the project "Usine Logicielle du pôle System@tic Paris-Région." This paper has won the SSTIC 2007 Best Technical Paper Prize.

X. Allamigeon (✉) · C. Hymans
EADS Innovation Works, SE/CS–Suresnes, France
e-mail: Xavier.Allamigeon@eads.net

X. Allamigeon
CEA, LIST MeASI, Gif-sur-Yvette, France

[1] In other words, all the existing errors are detected: there is no false negative.

| Year | Operating system | Lines of code (in million) |
|---|---|---|
| 1993 | Windows NT 3.1 | 6 |
| 1994 | Windows NT 3.5 | 10 |
| 1996 | Windows NT 4.0 | 16 |
| 2000 | Windows 2000 | 29 |
| 2001 | Windows XP | 40 |
| 2005 | Windows Vista Beta 2 | 50 |

**Fig. 1** Evolution of the size of the code of Microsoft Windows [47]

In this paper, we present a static analysis by abstract interpretation which can show the absence of heap overflows. In particular, the analysis is applied to the program given in Fig. 2, which will be used as an illustration throughout the paper. First, the language handled by the analysis is discussed in Sect. 2. Then, the semantics of this language, i.e. the mathematical model of the behavior of each construct, is formalized in Sect. 3. In Sect. 4, the principle of abstraction interpretation and the nature of the over-approximation performed on the semantics are explained. Finally, some possible extensions for the analysis are proposed in Sect. 5, while related work with respect to static analysis is discussed in Sect. 6.

## 2 Language and syntax

Static analysis of software written in a real (high-level) programming language, like C, is hard because of the language complexity and richness. Indeed, the C language has many diverse constructs, some with implicit semantics information, some with unspecified or architecture and compiler-dependent behaviors, some other being redundant. This is the reason why, for sake of concision, we restrict ourselves to a simpler kernel language that nevertheless embodies C essential paradigms. Moreover, as shown in Sect. 5, the analysis we describe here can be extended in numerous ways, and thus covers the whole C syntactic spectrum.

```
void readbuf(unsigned char* t, unsigned int n) {
    unsigned int i,sz;
    sz = (unsigned int) getchar();
    if (sz > n)
       sz = n;
    i = 0;
    while (i < sz) {
       *(t+i) = (unsigned char) getchar();
       i++;
    }
}

unsigned char* receive(unsigned int n) {
    assert (n>0);
    unsigned char* t = (unsigned char*)malloc(n);
    readbuf(t,n);
    return t;
}
```

**Fig. 2** A program in the C programming language to be analyzed. The function `receive` first makes sure that the value of `n` is strictly positive, and then allocates a buffer `t` of `n` characters in the heap. After that, it calls the function `readbuf` so as to collect in this buffer the data coming from an external source, by calling the function `getchar`. The latter reads for example the standard input or redirects data from a network. This program do not cause any heap overflow, because the variable `i` keeps a value which is smaller than `n` when dereferencing `*(t+i)`

Translation from the C language to our kernel language is not explained here, as it is fairly obvious (although quite tedious). Some intermediate forms such as [37] and especially [29] and their associated compilers are good starting points for this translation.

In the following section, we describe the fundamental principles of the kernel language we analyze and then we define its syntax.

### 2.1 Principles of the language

The only integer type in the language is the **unsigned char** type (shortened **uchar**), and the only possible pointers have the **uchar**$*$ type. Therefore, only one level of dereferencing is allowed. Moreover, type castings are forbidden. As a result, we assume that the type of each variable is (statically) known, and that the set of variables of **uchar** type and the set of pointers are disjoint. The (finite) set of all variables, and the (disjoint) sets of variables of **uchar** and **uchar**$*$ types are respectively denoted by Vars, CharVars, and PtrVars.

We suppose that there exists a function getuchar that returns a random unsigned character whenever it is called. It behaves in the same way as the function `getchar` in the C language: the values it returns are arbitrary ones as they are read from an external source (standard input, file, network, etc.), potentially controlled by the attacker.

Following standard C compilation conventions, the memory is split in two disjoint parts: the stack and the heap. They respectively contain statically allocated data (i.e. by variable declaration) and dynamically allocated data (i.e. by call to function malloc). We assume that the values located in the heap are all of **uchar** type, thus assimilating the heap to a memory area to store strings. Newly allocated data start with arbitrary values (this is slightly different from the C language where global variables are, by default, initialized to 0 with most compilers).

Pointers can only refer to addresses in the heap. Consequently, the stack cannot be written to through pointers. This is syntactically ensured by the absence of the & operator in the language. For sake of simplicity, the value **null** of pointers is not considered (although it can be easily added, as explained in Sect. 5).

Finally, function definitions or calls (except for malloc and getuchar) are not considered.

$$
\begin{array}{rll}
cmd & ::= instr_{stack} & \text{stack instruction} \\
& \mid instr_{heap} & \text{heap instruction} \\
& \mid cmd_1; cmd_2 & \text{sequence of commands} \\
& \mid \mathsf{while}\ cond\ \{\ cmd\ \} & \text{loop} \\
& \mid \mathsf{if}\ cond\ \{\ cmd_1\ \}\ \mathsf{else}\ \{\ cmd_2\ \} & \text{conditional} \\
\\
instr_{stack} & ::= x = e & \text{arithmetic assignment} \\
& \mid x = \mathsf{getuchar}() & \text{random character assignment} \\
& \mid p = q + e & \text{pointer assignment} \\
& \mid p = \mathsf{malloc}_\alpha(e) & \text{heap allocation} \\
\\
instr_{heap} & ::= *p = e & \text{heap assignment} \\
\\
cond & ::= e\ (\leq \mid ==)\ 0 & \text{comparison with 0} \\
& \mid\ !cond & \text{negation} \\
\\
e & ::= c & \text{character constant} \\
& \mid x & \text{variable of type } \mathbf{uchar} \\
& \mid \mathsf{op}(e_1, e_2) & \text{binary operation}
\end{array}
$$

**Fig. 3** Syntax of the language (the symbol $p$ refers to a variable of pointer type whereas op describes the sum or the substraction of characters)

## 2.2 Syntax of the language

The syntax of the language is given in Fig. 3. A program consists of a sequence of commands. For sake of simplicity, variable declarations are implicit: all variables have a global scope and are created before the execution of the program.

The $instr_{stack}$ and $instr_{heap}$ commands which handle memory are called *instructions*. The remaining commands control the flow of execution. Each occurrence of the function malloc is distinguished thanks to a label $\alpha$, called the *allocation site*.[2] In a given program, no two call sites to malloc are labelled by the same allocation site symbol. The (finite) set of all allocation sites is denoted by Alloc.

*Example 1* The program $P_{receive}$ in Fig. 4 is a possible translation into our kernel language of the introductory example in Fig. 2. Let us notice how the initial assertion that $n$ is strictly greater than 0 has been removed. We assume it is made implicit by the subsequent call to malloc. Indeed, as explained more deeply in Sect. 3.3, any call to malloc with a negative or 0 argument is an error which immediately stops program execution. The code of $P_{receive}$ will be used as our working example to illustrate several points in the article.

# 3 Semantics of the language

In this section, we assign a formal semantics to programs written in the kernel language. We will define mathematical objects that unambiguously model memory states (Sect. 3.2), the behavior of each instruction (Sect. 3.3) and of a program (Sect. 3.4). Finally, the properties that must be verified in order to ensure the absence of any heap overflow during the execution will be formalized in Sect. 3.5.

[2] The need for such symbols is made clear in Sect. 3.2.

```
1 :   t = malloc_α(n);
2 :   sz = getuchar();
3 :   if (!(sz − n ≤ 0))
4 :     sz = n;
5 :   i = 0;
6 :   p = t;
7 :   while (i − sz + 1 ≤ 0) {
8 :     tmp = getuchar();
9 :     *p = tmp;
10 :    p = p + 1;
11 :    i = i + 1;
12 :  }
```

**Fig. 4** The program $P_{receive}$: a translation of the program given in Fig. 2 (the variables $i$, $n$, $sz$, and $tmp$ are of **uchar** type, and $p$ and $t$ are pointers)

## 3.1 Some requirements

Let us first introduce some notations and notions that are required to define the formal semantics of the language.

**Control-flow graph** We suppose that any program $P$ written in the kernel language is provided as a *control-flow graph*. This directed graph describes the ordering of instructions and control instructions (condition). A node of this graph is a program control point.[3] An edge from $i$ to $j$ labelled by an instruction *instr* (or condition *cond*) in the graph depicts the possibility to go from the control point $i$ to the control point $j$ while executing the instruction *instr* (or alternatively if the condition *cond* is satisfied).

*Example 2* The control-flow graph of the program $P_{receive}$ is presented in Fig. 5.

The control-flow graph can be statically built from the source code of the program. This task is most classic, and we do not give further details here.

Given a program $P$, we denote by $\langle i, stmt, j\rangle$ the fact that there exists an edge from $i$ to $j$ labelled by *stmt* in the control-flow graph of $P$. The program $P$ is supposed to have an entry control point $entry(P)$, which is not reached by any incoming edge.[4]

**Transitions and inference rules** In what follows, the semantics of a program will be described by a relation $\rightarrow$, called *transition relation*. More precisely, we will have $\sigma_1 \rightarrow \sigma_2$ whenever the execution of an instruction or the application

[3] Provided that there no more than one instruction per line, a control point can be understood as the instruction line number.

[4] In C, this point typically corresponds to the *main* function.

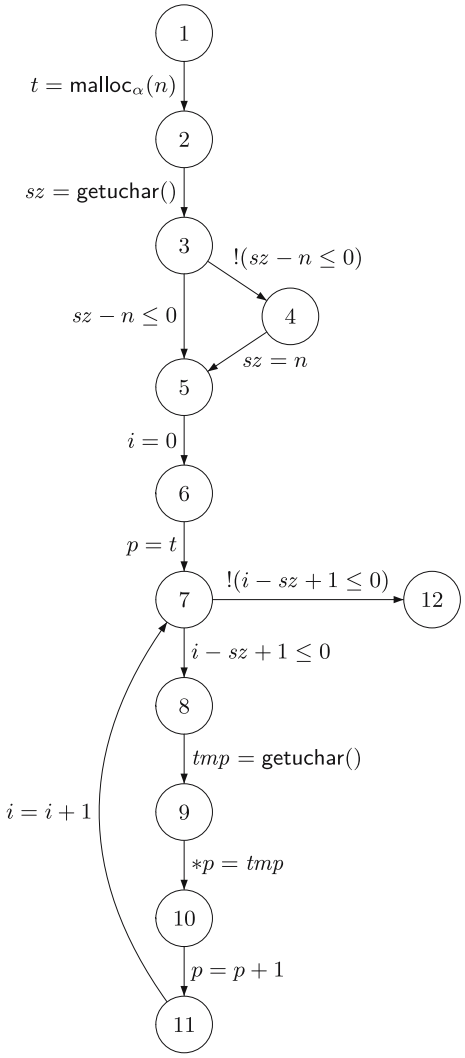



**Fig. 5** Control-flow graph of the program $P_{receive}$

of a condition starting from the machine state $\sigma_1$ results in the machine state $\sigma_2$.

The relation $\rightarrow$ will be defined by a set of *inference rules*. Each rule is of the form:

$$\frac{C_1 \quad \ldots \quad C_n}{D}$$

where the $C_i$ are called the *premises*, and $D$ the *conclusion*. Such a rule means: "if the $C_i$ are true, then $D$ is true".

### 3.2 Memory model

A memory state $s \restriction h$ is entirely described by its stack $s$ and its heap $h$. We denote the set of memory states by $\mathsf{Mem}$. The choices for the model of stack and heap, which are presented in the following parts, are summarized in Fig. 6.

**Stack model** A stack $s$ is modelled as a function that maps each program variable to a value whose nature depends on the variable type (character or pointer).

The set of characters is abusively assimilated to $\mathbb{N}$ (set of natural integers). Such a model is valid only in the absence of integer overflows, when all C language arithmetic operations behave as if they manipulate natural integers. However, this restriction could be easily dropped (see Sect. 5).

The value of a pointer is either the special value $\omega$, meaning that the pointer is not initialized, or an address in memory. The set of all addresses is denoted by $\mathsf{Addr}$.[5]

To summarize, the stack $s$ is a function which maps the set of variables in $\mathsf{Vars}$ to the set of values $\mathbb{N} \cup (\{\omega\} \cup \mathsf{Addr})$. The set of all possible stacks is denoted by $\mathsf{Stack}$.

**Model of the heap** Similarly, the heap $h$ maps addresses in $\mathsf{Addr}$ to characters in $\mathbb{N}$. However, the heap is defined only for the addresses that have been previously allocated by some call to $\mathsf{malloc}$. It is therefore a *partial* function, whose domain is denoted by $\mathsf{dom}(h)$, and any attempt to write to an address that is not allocated (not in $\mathsf{dom}(h)$), causes a heap overflow (this is explained further in Sect. 3.5). The set of all heaps is denoted by $\mathsf{Heap}$.

**Model of addresses** On a real machine, a memory address amounts to some integer value. However, the value of addresses is somewhat arbitrary: it can not be deduced from the program listing, and can changed from one execution to another. We prefer to model addresses in a more abstract way. What really matters is the fact, that each memory block created by a call to $\mathsf{malloc}$ is a sequence of consecutive bytes that does not overlap any previously allocated block.

In order to ensure the absence of heap overflows, it suffices to show that every memory write accesses is done within the bounds of these allocated blocks. Any access outside these blocks results in an error that terminates execution in our model (see Sect. 3.5). Thus, it is not necessary to know the exact position of these blocks in memory. Nevertheless, arithmetic of pointers within a given memory block must still make sense.[6]

To satisfy these constraints, we describe an address $a$ by a pair $(l_\alpha, o)$, of a *location* $l_\alpha$ and an *offset* $o$. The location $l_\alpha$ designates the memory block allocated by a call to $\mathsf{malloc}_\alpha$ at allocation site $\alpha$. That is the only thing we know about this memory block. In particular, the exact position of the block in memory is unknown. The offset $o$ is a positive integer. It describes the distance in bytes from the beginning of block $l_\alpha$. The offset allows pointer arithmetic in our model. For instance, $(l_\alpha, o-1)$ is the address located just before $(l_\alpha, o)$,

[5] Obviously, $\omega \notin \mathsf{Addr}$.

[6] Indeed, it is the only type of pointer arithmetic whose behavior is defined in the ANSI C standard [23, Sect. 6.5.6, Paragraph 8].

**Fig. 6** Summary of the choices to model memory states (for each set, the nature of its elements is given below)

$\mathsf{States} \stackrel{def}{=} \mathsf{Stack} \times \mathsf{Heap}$
$\sigma = s \mid h$ verifying (1)

$\mathsf{Stack} \stackrel{def}{=} (\mathbb{N} \cup \{\omega\} \cup \mathsf{Addr})^{\mathsf{Vars}}$
$s$ function from $\mathsf{Vars}$ to $(\mathbb{N} \cup \{\omega\} \cup \mathsf{Addr})$

$\mathsf{Heap} \stackrel{def}{=} \mathbb{N}^{(\mathsf{Addr})}$
$h$ partial function from $\mathsf{Addr}$ to $\mathbb{N}$

$\mathsf{Addr} \stackrel{def}{=} \mathsf{Loc} \times \mathbb{N}$
$a = (l_\alpha, o)$

and $(l_\alpha, o+1)$ the one just after. Similarly $(l_\alpha, 0)$ represents the first address of the block $l_\alpha$.

The set of locations will be denoted by $\mathsf{Loc}$. By definition, the set of addresses is thus $\mathsf{Addr} \stackrel{def}{=} \mathsf{Loc} \times \mathbb{N}$. We sometimes abusively write $l_\alpha \in \mathsf{dom}(h)$ to denote the fact that there exists $o \in \mathbb{N}$ such that $(l_\alpha, o) \in \mathsf{dom}(h)$.

**Well-formed memory states** Every memory state $s \mid h$ must verify an additional invariant to be valid:

$$\forall l_\alpha \in \mathsf{Alloc}. \forall p \in \mathsf{PtrVars}. \text{ if } s(p) = (l_\alpha, \cdot), \text{ then } l_\alpha \in \mathsf{dom}(h). \tag{1}$$

In other words, a pointer cannot point to a block that does not exist in the heap.[7]

Such an invariant is preserved since, as mentionned in Sect. 3, the only possible way to introduce a new location $l_\alpha$ is to call the function $\mathsf{malloc}_\alpha(\cdot)$, which immediately allocates the corresponding block on the heap. Every other instruction trivially preserves the invariant.

*Example 3* Let us consider the program $P_{receive}$ given in Example 1. A possible memory state at control point 12 is $s_f \mid h_f$, where:

$$s_f : \begin{cases} i \mapsto 6 \quad p \mapsto (\lambda_\alpha, 6) \\ n \mapsto 8 \quad t \mapsto (\lambda_\alpha, 0) \\ sz \mapsto 6 \\ tmp \mapsto 0 \end{cases}$$

$$h_f : \begin{cases} (\lambda_\alpha, 0) \mapsto 72 & (\lambda_\alpha, 4) \mapsto 111 \\ (\lambda_\alpha, 1) \mapsto 101 & (\lambda_\alpha, 5) \mapsto 0 \\ (\lambda_\alpha, 2) \mapsto 108 & (\lambda_\alpha, 6) \mapsto 179 \\ (\lambda_\alpha, 3) \mapsto 108 & (\lambda_\alpha, 7) \mapsto 23 \end{cases}, \tag{2}$$

with $\lambda_\alpha \in \mathsf{Loc}$. The execution that leads to this state, makes the variable $n$ to be equal to 8. This implies that the size of each received message has to be less than eight characters. In this case, the message contains six characters and corresponds to the string "`Hello`" (with a null character at its end). The remaining part of the block contains arbitrary characters, that have not been overwritten since the call to malloc which has created this block.

[7] Even though, the offset of the pointer can still be outside the bounds of the block, in case of an overflow.

### 3.3 Operational semantics

**Semantic states** The semantic state $\sigma$ describes the current state of the machine. In our case, it consists of the current control point $i$ and a memory state $s \mid h$. If $\mathsf{Ctrl}$ corresponds to the set of the control points of the program, the set of semantic state is defined by $\mathsf{States} \stackrel{def}{=} \mathsf{Ctrl} \times \mathsf{Mem}$.

**Evaluation of expressions** We first define the evaluation of an expression $e$ (of **uchar** type) in a given memory state $s \mid h$. This evaluation returns a value $v \in \mathbb{N}$, which is denoted by $s \mid h \vdash e \Rightarrow v$. The evaluation is defined by induction on the form of the expression $e$:

$$\frac{c \in \mathbb{N}}{s \mid h \vdash c \Rightarrow c} \tag{3}$$

$$\frac{x \in \mathsf{CharVars}}{s \mid h \vdash x \Rightarrow s(x)} \tag{4}$$

$$\frac{s \mid h \vdash e_1 \Rightarrow v_1 \quad s \mid h \vdash e_2 \Rightarrow v_2 \quad \mathsf{op}(v_1, v_2) = v}{s \mid h \vdash \mathsf{op}(e_1, e_2) \Rightarrow v} \tag{5}$$

**Semantics of instructions** The semantics of instructions is given by the following rule: if *instr* is an instruction,

$$\frac{\langle i, instr, j \rangle \quad s \mid h \vdash instr : s' \mid h'}{(i, s \mid h) \rightarrow (j, s' \mid h')} \tag{6}$$

where the relation $s \mid h \vdash instr : s' \mid h'$, defined in Fig. 7, describes the side effect of the instruction *instr* on memory.

The interpretation of these rules is the following:

- the stack assignment $x = e$ consists in replacing in $s$ the value of $x$ by the result of the evaluation of $e$;
- the instruction $x = \mathsf{getuchar}()$ is a non-deterministic one: indeed, the function $\mathsf{getuchar}()$ can return any character $c$. This corresponds to a model that takes all the possible behaviors of the external source (including the hostile ones) into account;
- the pointer assignment $p = q + e$ replaces in $s$ the value of $p$ with the value of $q$ shifted by $e$ bytes. If $q$ is not initialized, then $p$ has the value $\omega$;

**Fig. 7** Side effect of the instructions on memory (given a map $f$, $g \stackrel{def}{=} f[x \mapsto v]$ is the function which coincides with $f$ on its domain, except on $x$ where $g(x) = v$)

$$\frac{s \,¦\, h \vdash e \Rightarrow v}{s \,¦\, h \vdash x = e : s[x \mapsto v] \,¦\, h} \qquad \frac{c \in \mathbb{N}}{s \,¦\, h \vdash x = \mathsf{getuchar}() : s[x \mapsto c] \,¦\, h}$$

$$\frac{s(q) = (l_\alpha, o) \qquad s \,¦\, h \vdash e \Rightarrow v}{s \,¦\, h \vdash p = q + e : s[p \mapsto (l_\alpha, o + v)] \,¦\, h} \qquad \frac{s(q) = \omega}{s \,¦\, h \vdash p = q + e : s[p \mapsto \omega] \,¦\, h}$$

$$\frac{s \,¦\, h \vdash e \Rightarrow n \qquad n > 0 \qquad l_\alpha \notin \mathsf{dom}(h) \qquad w_0, \ldots, w_{n-1} \in \mathbb{N}}{s \,¦\, h \vdash p = \mathsf{malloc}_\alpha(e) : s[p \mapsto (l_\alpha, 0)] \,¦\, h[(l_\alpha, i) \mapsto w_i]_{0 \le i < n}}$$

$$\frac{\begin{array}{c} s \,¦\, h \vdash e \Rightarrow v \\ s(p) = (l_\alpha, o) \qquad (l_\alpha, o) \in \mathsf{dom}(h) \end{array}}{s \,¦\, h \vdash *p = e : s \,¦\, h[(l_\alpha, o) \mapsto v]}$$

- the instruction $p = \mathsf{malloc}_\alpha(e)$ introduces a new block $l_\alpha$ in the heap $h$. Thus, the latter is extended to the addresses $(l_\alpha, 0), \ldots, (l_\alpha, n-1)$, in which arbitrary characters $w_0, \ldots, w_{n-1}$ are put into (no initialization, as discussed in Sect. 2.1). Besides, the pointer $p$ now points to the beginning of the block, hence $p \mapsto (l_\alpha, 0)$ in the new stack. The location $l_\alpha$ has to be distinct from every other locations which already appears in the heap $h$: indeed, each call to malloc creates a new memory block which does not overlap the existing ones. Moreover, $n$ is required to be strictly positive. The case where $n$ is negative is considered as an error which stops execution (according to the standard ANSI C, the behavior of malloc(0) is implementation-dependent [23, Sect. 7.20.3, Paragraph 1]);
- finally, heap assignment $*p = e$ writes the value $v$ of $e$ to the address $s(p) = (l_\alpha, o)$ pointed to by $p$. Therefore, $p$ has to be initialized, and point to an address actually allocated in the memory (hence the requirement that $(l_\alpha, o) \in \mathsf{dom}(h)$).

**Semantics of conditions** The semantics of conditions is defined similarly to instructions: if *cond* is a condition,

$$\frac{\langle i, cond, j\rangle \qquad s \,¦\, h \vdash cond : s' \,¦\, h'}{(i, s \,¦\, h) \to (j, s' \,¦\, h')} \tag{7}$$

where the relation $s \,¦\, h \vdash cond : s' \,¦\, h'$, defined in Fig. 8, consists in selecting the memory states $s \,¦\, h$ which verify the condition *cond*. The state $s' \,¦\, h'$ is the same as initially, since the conditions do not have any side effect on memory.

$$\frac{s \,¦\, h \vdash e \Rightarrow 0}{s \,¦\, h \vdash (e == 0) : s \,¦\, h} \qquad \frac{s \,¦\, h \vdash e \Rightarrow v \qquad v \neq 0}{s \,¦\, h \vdash !(e == 0) : s \,¦\, h}$$

$$\frac{s \,¦\, h \vdash e \Rightarrow v \qquad v \le 0}{s \,¦\, h \vdash (e \le 0) : s \,¦\, h} \qquad \frac{s \,¦\, h \vdash e \Rightarrow v \qquad v > 0}{s \,¦\, h \vdash !(e \le 0) : s \,¦\, h}$$

**Fig. 8** Selection of memory states with respect to the conditions

*Example 4* Consider the program $P_{receive}$ whose control-flow graph is given in Fig. 5. The states $(1, s_1 \,¦\, h_1)$, $(2, s_2 \,¦\, h_2)$, $\ldots$, $(12, s_{12} \,¦\, h_{12})$ defined in Fig. 9 form a possible execution trace of the program:

$$(1, s_1 \,¦\, h_1) \to (2, s_2 \,¦\, h_2) \to \cdots \to (12, s_{12} \,¦\, h_{12}).$$

The final state $(12, s_{12} \,¦\, h_{12})$ coincides with the state given in Example 3, showing that the latter is effectively reachable (we will see in Sect. 3.4 that state $(1, s_1 \,¦\, h_1)$ is indeed a possible initial state).

### 3.4 Collecting semantics of a program

The collecting semantics $\mathcal{C}(P)$ of a program $P$ consists of all the machine states reachable during *any* execution of $P$. In particular, it is a subset of the set States. To define it precisely, let us introduce the function $F : \wp(\mathsf{States}) \to \wp(\mathsf{States})$ which corresponds to the execution of an additional step in the program:

$$F(X) \stackrel{def}{=} \left\{(entry(P), s_0 \,¦\, \emptyset) \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s_0(p) = \omega\right\} \cup \bigcup_{(i, s \,¦\, h) \in X} \left\{(j, s' \,¦\, h') \mid (i, s \,¦\, h) \to (j, s' \,¦\, h')\right\}. \tag{8}$$

Intuitively, the set $\left\{(entry(P), s_0 \,¦\, \emptyset) \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s_0(p) = \omega\right\}$ consists of the possible initial states of the machine: as mentionned in Sect. 2.1, the variables in the stack are not initialized,[8] and the heap does not contain any data. Besides, the set $\left\{(j, s' \,¦\, h') \mid (i, s \,¦\, h) \to (j, s' \,¦\, h')\right\}$ corresponds to the states reachable from states $(i, s \,¦\, h) \in X$ while executing an instruction or checking a condition.

Then, the collecting semantics $\mathcal{C}(P)$ is defined as the smallest solution of the equation $X = F(X)$. It is said to be the *smallest fixpoint* of $F$, which is denoted by $\mathcal{C}(P) \stackrel{def}{=} \mathsf{lfp}\, F$. Since $\wp(\mathsf{States})$ is a complete lattice and $F$ a monotone

[8] Thus, variables of **uchar** type may have any value in $\mathbb{N}$, and pointers take the value $\omega$.

$$\begin{cases} s_1 : i \mapsto 3, n \mapsto 8, sz \mapsto 53, tmp \mapsto 41, p \mapsto \omega, t \mapsto \omega, \\ h_1 : \end{cases}$$

$$\begin{cases} s_2 : i \mapsto 3, n \mapsto 8, sz \mapsto 53, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0), \\ h_2 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_3 : i \mapsto 3, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0) \\ h_3 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_5 : i \mapsto 3, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0) \\ h_5 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_6 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0) \\ h_6 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_7^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_7^0 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_8^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_8^0 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_9^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_9^0 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_{10}^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_{10}^0 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_{11}^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 1), t \mapsto (\lambda_\alpha, 0) \\ h_{11}^0 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_7^1 : i \mapsto 1, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 1), t \mapsto (\lambda_\alpha, 0) \\ h_7^1 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\vdots$$

$$\begin{cases} s_7^6 : i \mapsto 6, n \mapsto 8, sz \mapsto 6, tmp \mapsto 0, p \mapsto (\lambda_\alpha, 6), t \mapsto (\lambda_\alpha, 0) \\ h_7^6 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 101, (\lambda_\alpha, 2) \mapsto 108, (\lambda_\alpha, 3) \mapsto 108, \\ \quad (\lambda_\alpha, 4) \mapsto 111, (\lambda_\alpha, 5) \mapsto 0, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_{12} : i \mapsto 6, n \mapsto 8, sz \mapsto 6, tmp \mapsto 0, p \mapsto (\lambda_\alpha, 6), t \mapsto (\lambda_\alpha, 0) \\ h_{12} : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 101, (\lambda_\alpha, 2) \mapsto 108, (\lambda_\alpha, 3) \mapsto 108, \\ \quad (\lambda_\alpha, 4) \mapsto 111, (\lambda_\alpha, 5) \mapsto 0, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

**Fig. 9** A possible execution trace of the program $P$

map, the existence of such a fixpoint is ensured by Tarski's theorem [43].

As the function $F$ is semi-continuous, we know by a constructive version of Tarski's theorem [11], that:

$$\mathcal{C}(P) = \bigcup_{n \geq 0} F^n(\emptyset), \tag{9}$$

where $F^n$ is the $n$th iterate of $F$. Intuitively, this means that the collecting semantics consists of all the states reachable after a finite number of execution steps.

An equivalent definition is to assimilate $\mathcal{C}(P)$ to a function which maps the set $\mathsf{Ctrl}$ to the set $\wp(\mathsf{Mem})$, with each $i \in \mathsf{Ctrl}$ being mapped to the set of memory states possibly arising at the control point $i$. Then, $F$ can be redefined as the application which maps each $X : \mathsf{Ctrl} \to \wp(\mathsf{Mem})$ to the function $F(X) : \mathsf{Ctrl} \to \wp(\mathsf{Mem})$ defined by: for each $j \in \mathsf{Ctrl}$,

$$F(X)(j) \stackrel{def}{=} \begin{cases} \{s_0 \,|\, \emptyset \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s_0(p) = \omega\} \\ \qquad \text{if } j = entry(P) \\ \bigcup_{s \,|\, h \in X(i)} \{s' \,|\, h' \mid (i, s \,|\, h) \to (j, s' \,|\, h')\} \\ \qquad \text{otherwise} \end{cases} . \tag{10}$$

And we still have $\mathcal{C}(P) = \mathsf{lfp}\, F$, and

$$\mathcal{C}(P) = \bigcup_{n \geq 0} F^n(\emptyset). \tag{11}$$

With a slight abuse in notation, these definitions of the collecting semantics will be used indifferently throughout the paper.

### 3.5 Proving the absence of heap overflows

We now define the properties to be verified by a program to ensure the absence of heap overflow during the execution.

A heap overflow occurs if the program writes to an address not allocated in the heap. In our formalism, if $(i, s \,|\, h) \in \mathcal{C}(P)$ and $\langle i, *p = e, j \rangle$, there is a heap overflow if and only if $s(p) = \omega$ or $s(p) = (l_\alpha, o)$ with $(l_\alpha, o) \notin \mathsf{dom}(h)$.

Heap overflows can be modelled by an *error state* $\square$, added to the set $\mathsf{States}$, and the definition of the inference rules:

$$\frac{\langle i, *p = e, j \rangle \quad s(p) = \omega}{(i, s \,|\, h) \to \square} \tag{12}$$

$$\frac{\langle i, *p = e, j \rangle \quad s(p) = (l_\alpha, o) \notin \mathsf{dom}(h)}{(i, s \,|\, h) \to \square} \tag{13}$$

Then, the absence of heap overflow is equivalent to the condition $\square \notin \mathcal{C}(P)$. Nethertheless, this choice leads to a more complex formalism in the following parts. For that reason, we prefer encoding the error state by the absence of transition, as if the machine stops whenever the program tries to write to an invalid part of the memory. Then, the error state is assimilated to an unreachable state. And to ensure that a program does not cause any heap overflow, it suffices that any state $(i, s \,|\, h) \in \mathcal{C}(P)$ such that $\langle i, *p = e, j \rangle$ verifies:

$$\exists (l_\alpha, o) \in \mathsf{Addr}.s(p) = (l_\alpha, o) \in \mathsf{dom}(h). \tag{14}$$

### 3.6 Towards the analysis

Our goal is now to automatically check that every of the states that is reachable during the execution of the program to be analyzed verifies Property (14). Naively, we could proceed as follows: given a program $P$, "compute" the collecting

semantics $\mathcal{C}(P)$ and check whether each state satisfies Property (14). Naturally, $\mathcal{C}(P)$ is not computable by Rice's theorem [39], that implies the undecidability of the property in our language: an algorithm that takes any program as input and always decides in finite time whether there can be a heap overflow, does not exist.

More precisely, several difficulties would have to be tackled to compute $\mathcal{C}(P)$ for any $P$: (i) the subsets of $\wp(\mathsf{States})$ are not representable in machine, i.e. in general, they can not be stored as finite structures in a computer; (ii) the function $F$ is not computable (intuitively, it cannot be implemented as an algorithm); (iii) and the iteration sequence to compute $\mathcal{C}(P)$ by iterations (Eq. (9)) may not necessarily converge after a finite number of steps.

Fortunately, performing approximations on both semantics states and transfer functions can solve all the problems above-mentioned. Since we wish to ensure the *absence* of heap overflow at execution, these approximations should not forget any reachable state. Therefore, it is necessary to use *over-approximations*. It yields an algorithm which returns two possible results: either "it is certain that the program does not produce any heap overflow at execution", or "I don't know".[9] Abstract interpretation is based on this principle.

## 4 Abstract interpretation

In this section, we describe the approach discussed in Sect. 3.6 to solve the problem by approximation. It is first illustrated by an introductive example of abstraction in Sect. 4.1, thus which introduces the formalism defined in Sect. 4.2. A numerical abstraction by convex polyhedra is then discussed in Sect. 4.3. Finally, the abstraction used to analyze programs in the language of Sect. 2 is presented in Sect. 4.4.

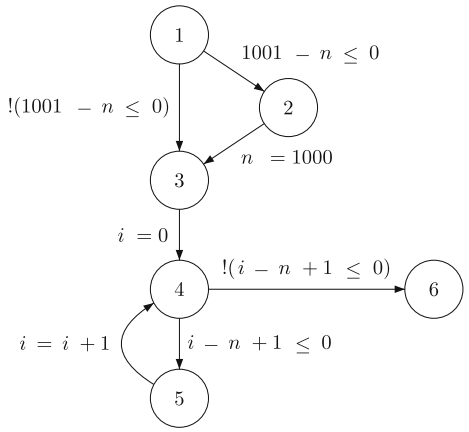
### 4.1 Introductive example: abstraction by intervals

We informally explain the computation of an over-approximation of the collecting semantics by using integer intervals. This abstraction has been first introduced in [10]. For sake of simplicity, we restrict ourselves to programs without pointers. Therefore, the memory state is reduced to a stack $s : \mathsf{Vars} \to \mathbb{N}$. As an illustrative example, let us consider the following program:

```
1 :  if (n > 1000)
2 :    n = 1000;
3 :  i = 0;
4 :  while (i < n) {
5 :    i = i + 1;
6 :  }
```

[9] Nevertheless, the algorithm could signal some overflows which are bound to occur. This feature is not presented here.

which normalizes $n$ to a value necessarily less than 1000, and then increments $i$ to $n$. Its control-flow graph is:




Integer intervals allow to over-approximate any subset $X$ of $\mathbb{R}$: if $l$ and $u$ are respectively the lower and upper bounds of $X$ in $\mathbb{R} \cup \{-\infty, +\infty\}$, $X$ is included in the interval $[l; u]$ (intervals may be open if $l$ or $u$ are equal to $\pm\infty$). Note that such an interval is computer-representable.[10]

We are now going to show informally how to use this elementary abstraction to compute an over-approximation of the collecting semantics. More precisely, we start from Eq. (11), and compute an over-approximation of the sets $F^n(\emptyset)$, where $F$ is defined as in Equation (10). Each $F^n(\emptyset)$ collects the set of possible memory states after $n-1$ execution steps for each control point. Our approximation of $F^n(\emptyset)$ consists in collecting an "abstract" state of memory mapping each program variable to an interval which over-approximates the values arising in all the real executions, instead of sets of memory states. This provides a computer-representable abstraction of $F^n(\emptyset)$.

*Example 5* If the following states are collected at a given control point:

$$\begin{cases} s_1 : i \mapsto 10, & n \mapsto 0 \\ s_2 : i \mapsto 1, & n \mapsto 127 \\ s_3 : i \mapsto 7, & n \mapsto 132 \end{cases}$$

then the set $\{s_1, s_2, s_3\}$ can be over-approximated by using intervals, by:

$$\mathcal{S} : i \mapsto [1; 10], \quad n \mapsto [0; 132].$$

The first iteration step $F(\emptyset)$ corresponds to the initial memory states, before the program is executed. As $i$ and

[10] In practice, integers to be over-approximated are at most encoded on 32 (or 64) bits. Therefore, intervals may be represented by a couple of integers of closed size. In particular, it is not necessary to represent neither large intervals nor $\pm\infty$.

$n$ are both unsigned characters, they can be abstracted to:

$$\mathcal{X}_1 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 3: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 4: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 5: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 6: & n \mapsto \emptyset, & i \mapsto \emptyset \end{cases}$$

where $\emptyset$ designates an empty interval. The application of the condition $1001 - n \leq 0$ and its negation yield the state:

$$\mathcal{X}_2 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 5: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 6: & n \mapsto \emptyset, & i \mapsto \emptyset \end{cases}$$

since at the control points 2 and 3, only memory states $s$ such that $s(n) > 1000$ and $s(n) \leq 1000$ respectively, can be reached. Similarly, the next step propagates the abstractions:

$$\mathcal{X}_3 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 0] \\ 5: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 6: & n \mapsto \emptyset, & i \mapsto \emptyset \end{cases}.$$

Then, the conditions $i + n - 1 \leq 0$ and $!(i + n - 1 \leq 0)$ with respect to the state are applied, returning at the control point 5 the states such that $n > 0$, and at 6 those verifying $n \leq 0$:

$$\mathcal{X}_4 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 0] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 0] \\ 6: & n \mapsto [0; 0]\,, & i \mapsto [0; 0] \end{cases}.$$

The incrementation of $i$ from the point 5 and to the point 4 yields a state in which the new value of $i$ is either equal to 0 (its old valued), or to 1. So we have:

$$\mathcal{X}_5 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 0] \\ 6: & n \mapsto [0; 0]\,, & i \mapsto [0; 0] \end{cases},$$

and then:

$$\mathcal{X}_6 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 1] \\ 6: & n \mapsto [0; 1]\,, & i \mapsto [0; 1] \end{cases},$$

And after 1998 further iterations, we finally get:

$$\mathcal{X}_{2004} = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 999] \\ 6: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \end{cases}.$$

which corresponds to a fixpoint. Indeed, a further iteration would yield the same result.

As a result, an over-approximation of the least fixed point of $F$, i.e. of $\mathcal{C}(P)$, has been obtained in a finite number of steps. And even if the computation has not exactly been made explicit, there exists an algorithm for computing automatically $\mathcal{X}_n$.

According to the final abstraction $\mathcal{X}_{2004}$, it is sure that the value of $i$ is bounded by 0 and 1000, whatever the initial value of $n$ may be. But, in counterpart, the abstraction by intervals is not precise enough to show that $i = n$.

Besides, the reader may be surprised by the large number of iterations necessary to reach a fixpoint. However, less precise approximations could have been performed for $\mathcal{X}_5$, yielding rather:

$$\mathcal{X}'_5 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 0] \\ 6: & n \mapsto [0; 0]\,, & i \mapsto [0; 0] \end{cases}.$$

Then, the propagation would have returned:

$$\mathcal{X}'_6 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[ \\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[ \\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[ \\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 999] \\ 6: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \end{cases},$$

so that, after 6 steps, the same fixpoint would have been reached. The operation which provides a less precise approximation is called *widening*. It does not only reduce the number of computation steps, but it also ensures this set to be finite. This notion will be further explained in the next section, in which the principles of abstraction are formalized.

### 4.2 Formalizing abstraction

A possible mathematical model for abstract interpretation is presented here. Other formalisms are possible [12].

**Principles** Let $D$ be a set containing the elements to be abstracted. We assume that $D$ is a complete lattice $(D, \preceq, \bot, \top, \curlyvee, \curlywedge)$: $\preceq$ is a partial order on the elements of $D$, $\bot$ and $\top$ are respectively the least and greatest elements of $D$, and $\curlyvee$ and $\curlywedge$ are union and intersection operators.[11] The set $D$ is called the *concrete domain*. It contains *concrete* elements, as opposed to abstract ones.

The abstraction is defined by an *abstract domain* $\mathcal{D}$, provided with a partial order $\sqsubseteq$, and a *concretization operator* $\gamma : \mathcal{D} \to D$ which is monotone, i.e.:

$$\forall \mathcal{X}, \mathcal{Y} \in \mathcal{D}.\ \text{if } \mathcal{X} \sqsubseteq \mathcal{Y}, \text{ then } \gamma(\mathcal{X}) \preceq \gamma(\mathcal{Y}). \tag{15}$$

The set $\mathcal{D}$ consists of abstract elements, and each element $\mathcal{X} \in \mathcal{D}$ corresponds to a concrete one: $\gamma(\mathcal{X})$. The ordering $\sqsubseteq$ is related to the precision of the abstraction: if $\mathcal{X} \sqsubseteq \mathcal{Y}$, then $\mathcal{X}$ is more precise than $\mathcal{Y}$, since it represents a smaller concrete element (for the order $\preceq$).

*Example 6* In the example given in Sect. 4.1, the concrete domain $D$ is the set $\wp(\mathsf{States})$, provided with the inclusion order $\subseteq$, and the abstract domain is defined by:

$$\mathcal{D} = \left(\mathcal{R}(\mathbb{Z})^{\mathsf{Vars}}\right)^{\mathsf{Ctrl}}$$

where $\mathcal{R}(\mathbb{Z})$ is the set of intervals over $\mathbb{Z}$, i.e.:

$$\begin{aligned}\mathcal{R}(\mathbb{Z}) = \{\emptyset\} &\cup \{[l; u] \mid l, u \in \mathbb{Z} \wedge l < u\} \\ &\cup \{]-\infty; u] \mid u \in \mathbb{Z}\} \cup \{[l; +\infty[ \mid l \in \mathbb{Z}\} \\ &\cup \{]-\infty; +\infty[\}\,.\end{aligned}$$

In other words, an element $\mathcal{X} \in \mathcal{D}$ maps each control point $i$ to an abstract memory state $\mathcal{X}(i)$, which is an application from $\mathsf{Vars}$ to $\mathcal{R}(\mathbb{Z})$. The partial ordering $\sqsubseteq$ on $\mathcal{D}$ is then defined by:

$$\mathcal{X} \sqsubseteq \mathcal{Y} \text{ if } \forall i \in \mathsf{Ctrl}. \forall x \in \mathsf{Vars}. \mathcal{X}(i)(x) \subseteq \mathcal{Y}(i)(x).$$

and the concretization $\gamma$ by:

$$\begin{aligned}\gamma(\mathcal{X}) : \mathsf{Ctrl} &\to \wp(\mathsf{Mem}) \\ i &\mapsto \{s \mid \forall x \in \mathsf{Vars}. s(x) \in \mathcal{X}(i)(x)\}.\end{aligned}$$

**Sound abstract operators** Given a function $F$ which maps $D$ to itself, we are interested in defining an abstract "equivalent" of $F$ on $\mathcal{D}$. This equivalent is required to be *sound*, i.e. to correspond to an "over-approximation" of $F$. Formally, if $\mathcal{F} : \mathcal{D} \to \mathcal{D}$, $\mathcal{F}$ is said to be *sound* w.r.t. $F$ if the following condition holds:

$$\forall \mathcal{X} \in \mathcal{D}. F(\gamma(\mathcal{X})) \preceq \gamma(\mathcal{F}(\mathcal{X})). \tag{16}$$

In other words, starting from the over-approximation $\mathcal{X}$ of $\gamma(\mathcal{X})$, $\mathcal{F}(\mathcal{X})$ is still an over-approximation of $F(\gamma(\mathcal{X}))$. Then, the application of $\mathcal{F}$ on $\mathcal{X}$ does not miss any concrete elements.

This definition can be generalized to introduce sound abstract operators of usual operations or elements of $D$. For instance,

- for the union $\curlyvee$, we will suppose that there exists an operator $\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ such that:

$$\gamma(\mathcal{X}) \curlyvee \gamma(\mathcal{Y}) \preceq \gamma(\mathcal{X} \sqcup \mathcal{Y}); \tag{17}$$

- similarly, for the intersection $\curlywedge$ we define an abstract operator $\sqcap$ verifying:

$$\gamma(\mathcal{X}) \curlywedge \gamma(\mathcal{Y}) \preceq \gamma(\mathcal{X} \sqcap \mathcal{Y}); \tag{18}$$

- and finally, we will assume that $\mathcal{D}$ is provided with a least element $\bot$ and a greatest element $\top$ (with respect to the ordering $\sqsubseteq$), corresponding to over-approximations of $\bot$ and $\top$ respectively.

*Example 7* The concrete domain $D = \wp(\mathsf{States})$ is a complete lattice, thus it is provided with union and intersection operators ($\cup$ and $\cap$ respectively), and with a least element, $\emptyset$, and a greatest one, $\mathsf{States}$. Sound abstract operators can be defined by: for any $i \in \mathsf{Ctrl}$,

$$\begin{aligned}(\mathcal{X} \sqcup \mathcal{Y})(i) &\overset{def}{=} x \mapsto \mathcal{X}(i)(x) \bowtie \mathcal{Y}(i)(x), \\ (\mathcal{X} \sqcap \mathcal{Y})(i) &\overset{def}{=} x \mapsto \mathcal{X}(i)(x) \cap \mathcal{Y}(i)(x), \\ \bot(i) &\overset{def}{=} x \mapsto \emptyset, \\ \top(i) &\overset{def}{=} x \mapsto ]-\infty; +\infty[\end{aligned}$$

where $\mathcal{X}(i) \bowtie \mathcal{Y}(i)$ is the smallest interval containing both $\mathcal{X}(i)$ and $\mathcal{Y}(i)$, and where $x \mapsto f(x)$ represents the function mapping each $x$ to $f(x)$.

**Abstract fixpoint computation** Let us suppose that $F$ is a monotone function on the complete lattice $(D, \preceq, \bot, \top, \curlyvee, \curlywedge)$, and that for any increasing sequence $X_0, \ldots, X_n, \ldots$ of elements of $D$, we have $F\left(\curlyvee_n X_n\right) = \curlyvee_n F(X_n)$ ($F$ is said to be *semi-continuous*). Tarski's theorem [43] then

[11] In particular, for any $(X, Y) \in D^2$, $X \curlywedge Y \preceq X \preceq X \curlyvee Y$ and $X \curlywedge Y \preceq Y \preceq X \curlyvee Y$.

ensures that $F$ has a least fixpoint $F$, and by a constructive version of this theorem, [11], we have:

$$\mathsf{lfp}\, F = \bigvee_{n\geq 0} F^n(\dot{\perp}). \tag{19}$$

Let $\mathcal{F}$ be a sound and monotone abstraction of $F$. Then, each iterate $F^n(\dot{\perp})$ can be over-approximated by $\mathcal{F}^n(\perp)$. Indeed, $\dot{\perp} \preceq \gamma(\perp)$ by definition and by recursion, if for a given $n$, $F^n(\dot{\perp}) \preceq \gamma(\mathcal{F}^n(\perp))$, then

$$\begin{aligned}
&F^{n+1}(\dot{\perp}) = F(F^n(\dot{\perp}))\\
&\preceq F(\gamma(\mathcal{F}^n(\perp))) \text{ by recursion hypothesis and } F \text{ monotone}\\
&\preceq \gamma(\mathcal{F}(\mathcal{F}^n(\perp))) \qquad \text{by Eq. (16)}\\
&\preceq \gamma(\mathcal{F}^{n+1}(\perp)).
\end{aligned}$$

Moreover, the sequence of the $\mathcal{F}^n(\perp)$ is clearly monotone. Then, if this sequence is *stationary*, i.e. there exists an index $N$ from which all the terms of the sequence are equal (we will also say that the sequence converges in a finite number $N$ of steps), we have:

$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{F}^N(\perp)). \tag{20}$$

Such a situation occurred in the computation given in Sect. 4.1. But from a general point of view, the sequence may not be stationary. That is why we introduce a new operator $\nabla : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$, called *widening operator*, satisfying:

- $\forall \mathcal{X}, \mathcal{Y} \in \mathcal{D}. \mathcal{X} \sqcup \mathcal{Y} \sqsubseteq \mathcal{X} \nabla \mathcal{Y}$,
- for any increasing sequence of elements $\mathcal{X}_0 \preceq \cdots \preceq \mathcal{X}_n \preceq \ldots$, the sequence defined by

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \nabla \mathcal{X}_{n+1} \end{cases} \tag{21}$$

converges in a finite number $N$ of steps.

Then, the operator $\nabla$ allows to ensure the convergence of the sequence $\mathcal{F}^n(\perp)$ in a finite number of steps.

**Theorem 1** *If the sequence $(\mathcal{X}_n)_n$ is defined by:*

$$\begin{cases} \mathcal{X}_0 \stackrel{def}{=} \perp \\ \mathcal{X}_{n+1} \stackrel{def}{=} \begin{cases} \mathcal{X}_n & \text{if } \mathcal{F}(\mathcal{X}_n) \sqsubseteq \mathcal{X}_n \\ \mathcal{X}_n \nabla \mathcal{F}(\mathcal{X}_n) & \text{otherwise} \end{cases} \end{cases}, \tag{22}$$

*then this sequence is stationary, and its limit $\mathcal{X}_N$ verifies:*

$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{X}_N). \tag{23}$$

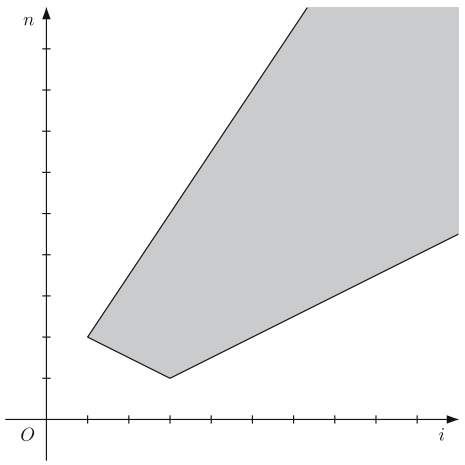
As a consequence, if the elements of the abstract domain $\mathcal{D}$ are computer-representable, if the function $\mathcal{F}$ is computable, and if $\mathcal{D}$ is provided with a widening operator $\nabla$ which is also computable, Theorem 1 yields an algorithm to compute an over-approximation of $\mathsf{lfp}\, F$ *in a finite amount of time*.

### 4.3 Convex polyhedra

In this section, we present a numeric abstract domain to approximate sets of tuples of integers by affine inequalities, i.e. a convex polyhedra. Such an abstraction allows to discover more precise invariants than intervals. It was first defined in [13].

Let $V = \{V_1, \ldots, V_d\}$ be a finite and ordered set of pairwise distinct variables. The set $\mathbb{CP}(V)$ consists of the convex polyhedra in the affine space $\mathbb{R}^d$, whose dimensions are labelled by the variables $V_1, \ldots, V_d$ respectively. A polyhedron $\mathcal{P} \in \mathbb{CP}(V)$ can be represented by a system of affine inequality constraints. For that reason, a polyhedron will be sometimes directly referred to by a constraint system.

*Example 8* For $d = 2$, let us consider the convex polyhedron over the variables $i$ and $n$, where $i$ and $n$ are respectively represented on the $X$- and $Y$-axes:




It is defined by the following constraint system:

$$\begin{cases} 2n - 3i \leq 1 \\ 2n + i \geq 5 \\ 2n - i \geq -1 \end{cases}.$$

The set $\mathbb{CP}(V)$ provided with the inclusion $\subseteq$ is a partially ordered set, and a concretization operator $\gamma_{poly} : \mathbb{CP}(V) \to \mathbb{R}^V$ can be defined by:

$$\gamma_{poly}(\mathcal{P}) = \{f : V \to \mathbb{R} \mid (f(V_1), \ldots, f(V_d)) \in \mathcal{P}\}. \tag{24}$$

Obviously, $\gamma_{poly}$ is monotone. Usual abstract operators can be also introduced:

– the union of two polyhedra $\mathcal{P}_1$ and $\mathcal{P}_2$ can be over-approximated by the convex hull $\mathcal{P}_1 \uplus \mathcal{P}_2$ of $\mathcal{P}_1 \cup \mathcal{P}_2$ (because $\mathcal{P}_1 \cup \mathcal{P}_2$ is not necessarily convex);
– the intersection of $\mathcal{P}_1$ and $\mathcal{P}_2$ can be abstracted by $\mathcal{P}_1 \cap \mathcal{P}_2$ (polyhedra are stable with respect to set intersection);
– the sets $\emptyset$ and $\mathbb{R}^d$ are the least and the greatest elements respectively.

Now some abstract transformations on polyhedra are presented. To be concise, we informally skim through them.

– The application on a polyhedron $\mathcal{P}$ of an (in)equality $c_0 + \sum_i c_i V_i \diamond 0$, where $\diamond \in \{\leq, =, \geq\}$, consists in intersecting $\mathcal{P}$ with the polyhedron defined by the (in)equality. If $(\!|c_0 + \sum_i c_i V_i \diamond 0|\!)(\mathcal{P})$ is the resulting polyhedron, we have:

$$\left\{ f : V \to \mathbb{R} \mid (f(V_1), \ldots, f(V_d)) \in \mathcal{P} \wedge \left( c_0 + \sum_i c_i f(V_i) \diamond 0 \right) \right\} \subseteq \gamma_{poly}\left( (\!| c_0 + \sum_i c_i V_i \diamond 0 |\!)(\mathcal{P}) \right). \tag{25}$$

– The assignment of a random value to a variable $V_i \leftarrow ?$ is defined as the convex hull of the union of the polyhedron $\mathcal{P}$ with the $i$th axis. Then:

$$\{ f : V \to \mathbb{R} \mid \exists v. (f(V_1), \ldots, f(V_{i-1}), v, f(V_{i+1}), \ldots, f(V_d)) \in \mathcal{P} \} \subseteq \gamma_{poly}\left( (\!| V_i \leftarrow ? |\!)(\mathcal{P}) \right). \tag{26}$$

– The assignment $V_i \leftarrow c_0 + \sum_j c_j V_j$ of an affine expression to a variable $V_i$ consists in:
  - when $c_i \neq 0$, replacing each occurrence of the variable $V_i$ by the expression $\frac{V_i - c_0 - \sum_{j \neq i} c_j V_j}{c_i}$ in the constraint system which defines $\mathcal{P}$;
  - when $c_i = 0$, assigning a random value to $V_i$, so as to "overwrite" its previous value, and then applying the condition $V_i - \left( c_0 + \sum_j c_j V_j \right) = 0$, to the resulting polyhedron.

  In both cases, this yields a polyhedron $(\!| V_i \leftarrow c_0 + \sum_j c_j V_j |\!)(\mathcal{P})$ which verifies:

$$\left\{ g : V \to \mathbb{R} \mid \begin{array}{c} (f(V_1), \ldots, f(V_d)) \in \mathcal{P} \wedge \forall j \neq i. g(V_j) = f(V_i) \\ \wedge\ g(V_i) = c_0 + \sum_j c_j f(V_j) \end{array} \right\} \subseteq \gamma_{poly}\left( (\!| V_i \leftarrow c_0 + \sum_j c_j V_j |\!)(\mathcal{P}) \right). \tag{27}$$

These transfer functions will be used to build a sound abstraction of the function $F$ introduced in Sect. 3.4, for any considered program.

Finally, a widening operator $\nabla_{poly}$ can be introduced: considering $\mathcal{P}, \mathcal{Q} \in \mathbb{CP}(V)$, the polyhedron $\mathcal{P} \nabla_{poly} \mathcal{Q}$ is defined by the following constraint system:

$$\begin{array}{l} \{ C \mid C \text{ constraint of } \mathcal{P} \text{ entirely satisfied in } \mathcal{Q} \} \\ \cup \{ C' \mid C' \text{ constraint of } \mathcal{Q} \text{ equivalent to a constraint of } \mathcal{P} \}. \end{array} \tag{28}$$

A constraint $C'$ is equivalent to a constraint of $\mathcal{P}$ if there exists a constraint $C''$ of $\mathcal{P}$ such that $\mathcal{P}$ remains unchanged when replacing $C''$ by $C'$:

$$\mathcal{P} = \left( \mathcal{P} \setminus \{C''\} \right) \cup \{C'\}.$$

The reader can refer to [13] for further explanations on $\nabla_{poly}$.

Obviously, elements of $\mathbb{CP}(V)$ are computer-representable (for instance, by using constraint systems), and the "geometric" operations involved in the definition of the abstract operators are computable as well. This abstraction, provided with the operator $\nabla_{poly}$, is thus well-suited for computing over-approximations. The polyhedron domain is also more precise than intervals. For instance, it is able to discover the invariant $i = n$ at the control point 6 in the program given in Sect. 4.1. Nevertheless, a drawback of this precision is the cost of the abstract operators: both in time and in memory, they have an exponential complexity in the number $d$ of variables. But other abstract domains, such as those described in [32,34,35,42], benefit a better trade-off between precision and complexity.

### 4.4 Final abstraction

We can now define an abstraction of $\wp(\mathsf{States})$ in order to analyze programs written in the language presented in Sect. 2.

Let us remind that $\mathsf{CharVars}$ and $\mathsf{PtrVars}$ are respectively the set of variables of **uchar** and pointer types, and $\mathsf{Alloc}$ the set of the allocation sites $\alpha$ labelling the instructions $p = \mathsf{malloc}_\alpha(e)$. For each pointer variable $p$, we introduce two variables, $p_\ell$ and $p_o$. Intuitively, they respectively represent the location and the offset of the address pointed to by $p$. The sets of the $p_\ell$ and the $p_o$ are denoted by $\mathsf{LocVars}$ and $\mathsf{OffVars}$ respectively. Moreover, for every symbol $\alpha$ labelling an instruction $p = \mathsf{malloc}_\alpha(e)$, a variable $\alpha_s$ corresponding to the size of the memory block allocated at the site $\alpha$, is defined. Variables $\alpha_s$ form the set $\mathsf{SizeVars}$. Finally, as the variables in $\mathsf{CharVars}$, $\mathsf{OffVars}$, and $\mathsf{SizeVars}$ are all numerical ones, they are merged in a set denoted by $\mathsf{NumVars}$. These notations are summarized in Fig. 10.

The abstract domain $\mathsf{AStates}$ contains the applications from $\mathsf{Ctrl}$ to $\mathsf{AMem}$, where $\mathsf{AMem}$ denotes the set of abstract memory states, and whose elements are pairs $(\mathcal{L}, \mathcal{N})$, where $\mathcal{L} : \mathsf{LocVars} \to \wp(\mathsf{Alloc} \cup \{\omega\})$ and $\mathcal{N} \in \mathbb{CP}(\mathsf{NumVars})$. In other words:

$$\begin{array}{l} \mathsf{AStates} \stackrel{def}{=} \mathsf{AMem}^{\mathsf{Ctrl}} \\ \mathsf{AMem} \stackrel{def}{=} \left( \wp(\mathsf{Alloc} \cup \{\omega\})^{\mathsf{LocVars}} \right) \mathbb{CP}(\mathsf{NumVars}). \end{array} \tag{29}$$

**Fig. 10** Symbolic and numerical variables

$$\mathsf{CharVars} \stackrel{def}{=} \{x \in \mathsf{Vars} \mid x \text{ of type } \mathbf{uchar}\} \quad \mathsf{PtrVars} \stackrel{def}{=} \{p \in \mathsf{Vars} \mid z \text{ of type } \mathbf{char}*\}$$
$$\mathsf{Alloc} \stackrel{def}{=} \{\alpha \mid p = \mathsf{malloc}_\alpha(e)\}$$
$$\mathsf{LocVars} \stackrel{def}{=} \{p_\ell \mid p \in \mathsf{PtrVars}\} \qquad \mathsf{OffVars} \stackrel{def}{=} \{p_o \mid p \in \mathsf{PtrVars}\}$$
$$\mathsf{SizeVars} \stackrel{def}{=} \{\alpha_s \mid \alpha \in \mathsf{Alloc}\} \qquad \mathsf{NumVars} \stackrel{def}{=} \mathsf{CharVars} \cup \mathsf{OffVars} \cup \mathsf{SizeVars}$$

Then, an abstract state maps each control point $i$ to an abstract memory representation $(\mathcal{L}_i, \mathcal{N}_i)$: (i) the function $\mathcal{L}_i$ maps each pointer to an abstract location, i.e. a subset of $\mathsf{Alloc} \cup \{\omega\}$, (ii) the polyhedron $\mathcal{N}_i$ is an over-approximation of the numerical variables. Moreover, the elements of $\mathsf{AStates}$ are effectively computer-representable since $\mathsf{Ctrl}$, $\mathsf{Alloc}$, $\mathsf{LocVars}$, and $\mathsf{NumVars}$ are all finite sets.

The partial ordering $\sqsubseteq$ on $\mathsf{AStates}$ is defined as the lift of the inclusion ordering $\subseteq$ on the domains $\wp(\mathsf{Alloc} \cup \{\omega\})$ and $\mathbb{CP}(\mathsf{NumVars})$, i.e.: $\mathcal{X} \sqsubseteq \mathcal{Y}$ if for any $i \in \mathsf{Ctrl}$,

$$\begin{cases} \forall p_\ell \in \mathsf{LocVars}. \mathcal{L}_i(p_\ell) \subseteq \mathcal{L}'_i(p_\ell) \\ \mathcal{N}_i \subseteq \mathcal{N}'_i \end{cases}, \tag{30}$$

where $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$ and $\mathcal{Y}(i) = (\mathcal{L}'_i, \mathcal{N}'_i)$.

The concretization operator $\gamma : \mathsf{AStates} \to \wp(\mathsf{States})$ is defined in the following way:

$$\gamma(\mathcal{X}) \stackrel{def}{=} \begin{cases} \mathsf{Ctrl} \to \wp(\mathsf{Mem}) \\ \quad i \mapsto \gamma_{mem}(\mathcal{X}(i)) \end{cases}, \tag{31}$$

where $\gamma_{mem} : \mathsf{AMem} \to \wp(\mathsf{Mem})$ gives the meaning of an abstract memory state $(\mathcal{L}, \mathcal{N})$:

$$\gamma_{mem}(\mathcal{L}, \mathcal{N}) \stackrel{def}{=}$$
$$\left\{ \begin{array}{ll} s \natural h & \in \mathsf{States} \wedge \nu \in \gamma_{poly}(\mathcal{N}) \\ & \wedge \forall x \in \mathsf{CharVars}. s(x) = \nu(x) \\ s \natural h \mid & \wedge \forall p \in \mathsf{PtrVars}. \left[ \begin{array}{l} (\alpha \in \mathcal{L}(p_\ell) \cap \mathsf{Alloc} \wedge s(p) = (l_\alpha, \nu(p_o))) \\ \vee (\omega \in \mathcal{L}(p_\ell) \wedge s(p) = \omega) \end{array} \right] \\ & \wedge \forall \alpha \in \mathsf{Alloc}. \forall l_\alpha \in \mathsf{dom}(h). \text{there exists } k_{l_\alpha} \text{ verifying} \\ & \big((l_\alpha, o) \in \mathsf{dom}(h) \Leftrightarrow 0 \le o < k_{l_\alpha}\big) \wedge \nu[\alpha_s \mapsto k_{l_\alpha}] \in \gamma_{poly}(\mathcal{N}) \end{array} \right\}. \tag{32}$$

Intuitively, for every control point $i \in \mathsf{Ctrl}$, if $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$,

- $\mathcal{L}_i$ over-approximates the locations of the memory blocks pointed to by pointers: $s(p)$ can either be under the form $(l_\alpha, \cdot)$ with $\alpha \in \mathcal{L}_i(p_\ell) \cap \mathsf{Alloc}$, or be uninitialized if $\omega \in \mathcal{L}_i(p_\ell)$;
- $\mathcal{N}_i$ over-approximated every numerical variables: the values of the characters $(s(x) = \nu(x))$, the offsets of the addresses pointed to by pointers $(s(p) = (\cdot, \nu(p_o)))$, and the size of the blocks in the heap: for any block of location $l_\alpha$ contained in the heap, its size $k_{l_\alpha}$ is over-approximated by $\alpha_s$ $(\nu[\alpha_s \mapsto k_{l_\alpha}] \in \gamma_{poly}(\mathcal{N}))$.

Thus, the key idea of the heap approximation is to keep some information regarding the size of the allocated blocks, while totally forgetting the exact content of the heap, and to merge the information concerning the blocks allocated at the same allocation site.

*Example 9* Let us consider the following abstract memory state:

$$(\mathcal{L}, \mathcal{N}) = \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \ge 1, tmp \ge 0, 0 \le p_o = i = sz \le \alpha_s = n\} \end{array} \right).$$

It represents all the memory states $s \natural h$ such that:

- $s(n) \ge 1, 0 \le s(i) \le n$, and $s(tmp) \ge 0$;
- $s(p) = (\lambda_\alpha, s(i))$ and $s(t) = (\lambda'_\alpha, 0)$, with $\lambda_\alpha, \lambda'_\alpha \in \mathsf{Loc}$ ($\lambda_\alpha$ and $\lambda'_\alpha$ may be equal);
- the heap $h$ contains at least two blocks $\lambda_\alpha$ and $\lambda'_\alpha$, which are possibly equal. Each of them has a size equal to $s(n)$. Indeed, the size $k_{\lambda_\alpha}$ of the block $\lambda_\alpha$, defined by $(\lambda_\alpha, o) \in \mathsf{dom}(h) \Leftrightarrow 0 \le o < k_{\lambda_\alpha}$, satisfies the equality $k_{\lambda_\alpha} = s(n)$. This is similar for the block $\lambda'_\alpha$.

Some usual abstract operators on $\mathsf{AStates}$ can be defined by using those defined on $\wp(\mathsf{Alloc} \cup \{\omega\})$ and $\mathbb{CP}(\mathsf{NumVars})$:

- a union abstract operator $\sqcup$, such that for any $i \in \mathsf{Ctrl}$, if $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$ and $\mathcal{Y}(i) = (\mathcal{L}'_i, \mathcal{N}'_i)$, then

$$(\mathcal{X} \sqcup \mathcal{Y})(i) \stackrel{def}{=} (\mathcal{L}_i, \mathcal{N}_i) \dot{\sqcup} (\mathcal{L}'_i, \mathcal{N}'_i), \tag{33}$$

  where $(\mathcal{L}, \mathcal{N}) \dot{\sqcup} (\mathcal{L}', \mathcal{N}')$ is itself an union abstract operator on $\mathsf{AMem}$:

$$(\mathcal{L}, \mathcal{N}) \dot{\sqcup} (\mathcal{L}', \mathcal{N}') \stackrel{def}{=} (p_\ell \mapsto \mathcal{L}(p_\ell) \cup \mathcal{L}'(p_\ell), \mathcal{N} \uplus \mathcal{N}'). \tag{34}$$

- the least element $\bot$, defined by $\forall i \in \mathsf{Ctrl}. \bot(i) \stackrel{def}{=} (p_\ell \mapsto \emptyset, \emptyset)$.
- a widening operator $\nabla$, where for any $i \in \mathsf{Ctrl}$, if $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$ and $\mathcal{Y}(i) = (\mathcal{L}'_i, \mathcal{N}'_i)$,

$$(\mathcal{X} \nabla \mathcal{Y})(i) \stackrel{def}{=} (p_\ell \mapsto \mathcal{L}_i(p_\ell) \cup \mathcal{L}'_i(p_\ell), \mathcal{N}_i \nabla_{poly} \mathcal{N}'_i)$$

Their soundness is based on the soundness of the underlying operators on the sets $\wp(\mathsf{Alloc} \cup \{\omega\})$ and $\mathbb{CP}(\mathsf{NumVars})$. Moreover, in the definition of the widening operator $\nabla$, the set union $\cup$ on the location component is sufficient to ensure the convergence in a finite number of iterations, because $\mathsf{Alloc} \cup \{\omega\}$ is a finite set, hence its height is finite. Other operators (intersection, greatest element) can be defined, but they are not required in the rest of the paper.

Now, given a program $P$, we are going to build a sound abstraction $\mathcal{F}$ of the transfer function $F$ defined by Equation (10): for each $\mathcal{X} \in \mathsf{AStates}$, and for any $j \in \mathsf{Ctrl}$, we define

$$\mathcal{F}(\mathcal{X})(j) \stackrel{def}{=} \begin{cases} \left(p_\ell \mapsto \{\omega\}, \bigcap_{x,\alpha_s} \{x \geq 0\} \cap \{\alpha_s = 0\}\right) & \text{if } j = entry(P) \\ \dot{\bigsqcup}_{\langle i, stmt, j\rangle} [\![stmt]\!](\mathcal{L}_i, \mathcal{N}_i) & \text{otherwise} \end{cases}, \tag{35}$$

where $[\![stmt]\!] : \mathsf{AMem} \to \mathsf{AMem}$ is a function defined below, and which verifies, for any instruction or condition *stmt*:

$$\begin{aligned} &\left\{s' \,\text{¦}\, h' \mid s \,\text{¦}\, h \in \gamma_{mem}(\mathcal{L}, \mathcal{N}) \wedge s \,\text{¦}\, h \vdash stmt : s' \,\text{¦}\, h'\right\} \\ &\quad \subseteq \gamma_{mem}\left([\![stmt]\!](\mathcal{L}, \mathcal{N})\right). \end{aligned} \tag{36}$$

In other words, $[\![stmt]\!]$ is required to be a sound abstraction of the effect of the instruction or of the condition *stmt* in the concrete semantics.

Obviously, the initial abstract state over-approximates the corresponding concrete state:

$$\begin{aligned} &\{s_0 \,\text{¦}\, \emptyset \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}. s(p) = \omega\} \\ &\qquad \subseteq \gamma_{mem}\left(p_\ell \mapsto \{\omega\}, \bigcap_{x,\alpha_s} \{x \geq 0\} \cap \{\alpha_s = 0\}\right), \end{aligned} \tag{37}$$

so that, if Relation 36 is satisfied, the following proposition holds:

**Proposition 1** *The abstract transfer function $\mathcal{F}$ is a sound abstraction of the concrete transfer function $F$.*

Finally, the functions $[\![stmt]\!]$ have to be precisely defined, using in particular abstract operators on polyhedra presented in Sect. 4.3.

$$[\![x = e]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|x \leftarrow e|\!)(\mathcal{N})) \tag{38}$$

$$[\![x = \mathsf{getuchar}()]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|x \geq 0|\!) \circ (\!|x \leftarrow ?|\!)(\mathcal{N})) \tag{39}$$

$$[\![p = q + e]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}[p_\ell \mapsto \mathcal{L}(q_\ell)], (\!|p_o \leftarrow q_o + e|\!)(\mathcal{N})) \tag{40}$$

$$\begin{aligned} &[\![p = \mathsf{malloc}_\alpha(e)]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}[p_\ell \mapsto \{\alpha\}], (\!|p_o \leftarrow 0|\!)(\mathcal{N}')) \\ &\text{where } \mathcal{N}' = (\!|\alpha_s \leftarrow e|\!) \circ (\!|e \geq 1|\!)(\mathcal{N}) \uplus (\!|\alpha_s \geq 1|\!)(\mathcal{N}) \end{aligned} \tag{41}$$

$$[\![*p = e]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}[p_\ell \mapsto \mathcal{L}(p_\ell) \cap \mathsf{Alloc}], \mathcal{N}) \tag{42}$$

$$[\![e \leq 0]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}, \,\text{¦}(e \leq 0\,\text{¦})(\mathcal{N})) \tag{43}$$

$$[\![e == 0]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}, \,\text{¦}(e = 0\,\text{¦})(\mathcal{N})) \tag{44}$$

$$[\![!(e \leq 0)]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}, \,\text{¦}(e \geq 1\,\text{¦})(\mathcal{N})) \tag{45}$$

$$[\![!(e == 0)]\!](\mathcal{L}, \mathcal{N}) \stackrel{def}{=} (\mathcal{L}, \,\text{¦}(e \leq -1\,\text{¦})(\mathcal{N}) \uplus \,\text{¦}(e \geq 1\,\text{¦})(\mathcal{N})) \tag{46}$$

In other words:

- assignments $x = e$, $x = \mathsf{getuchar}()$, and $p = q + e$ accordingly modify the abstract value of $x$ and $p_o$ respectively, in the polyhedron $\mathcal{N}$ thanks to the operator $\,\text{¦}(\cdot \leftarrow \cdot)\,\text{¦}$. Moreover, for pointer assignment, the abstract value of the location of $p$ is replaced by the location of the pointer $q$. The soundness property (36) directly results from the soundness of $\,\text{¦}(\cdot \leftarrow \cdot)\,\text{¦}$;
- dynamic allocation $p = \mathsf{malloc}_\alpha(e)$ updates the value of $\alpha_s$ with the abstract value of $e$ on which the condition $e \geq 1$ has been applied (this is the only case defined in the concrete semantics). Besides, the previous values of $\alpha_s$ has to be taken into account, since $\alpha$ may represent several distinct memory blocks.[12] Finally, the values $\{\alpha\}$ and 0 are assigned to $p_\ell$ and $p_o$ respectively, in order for the pointer $p$ to point at the beginning of the allocated block;
- heap assignment $*p = e$ excludes the case where $p$ is not initialized (hence $\mathcal{L}(p_\ell) \cap \mathsf{Alloc}$);
- finally, the conditions *cond* consists in applying the same condition on the numerical abstract values.

Given a program $P$, the abstract transfer function $\mathcal{F}$ is clearly computable. Then, Theorem 1 in Sect. 4.2 provides an algorithm to compute an over-approximation of the collecting semantics $\mathcal{C}(P)$ of $P$.

**Corollary 1** *If the sequence $(\mathcal{X}_n)_n$ is defined by:*

$$\begin{cases} \mathcal{X}_0 \stackrel{def}{=} \bot \\ \mathcal{X}_{n+1} \stackrel{def}{=} \begin{cases} \mathcal{X}_n & \text{if } \mathcal{F}(\mathcal{X}_n) \sqsubseteq \mathcal{X}_n \\ \mathcal{X}_n \nabla \mathcal{F}(\mathcal{X}_n) & \text{otherwise} \end{cases}, \end{cases} \tag{47}$$

*then this sequence is stationary. Its limit $\mathcal{X}_N$ is* computable, *and satisfies:*

$$\mathcal{C}(P) \subseteq \gamma(\mathcal{X}_N). \tag{48}$$

[12] Only the old values of $\alpha_s$ greater than or equal to 1 are to be considered, since allocated blocks necessarily have a strictly positive size.

**Absence of heap overflows** The computed over-approximation of the collecting semantics $\mathcal{C}(P)$ of a program $P$ allows to show the absence of heap overflows, when it is accurate enough. Let us consider the following property:

$$\forall i \in \mathsf{Ctrl}.\mathsf{if}\ \langle i, *p = e,\ j\rangle,\ \text{then } \mathcal{X}_N(i) = (\mathcal{L}_i, \mathcal{N}_i) \text{ verifies}$$
$$\omega \notin \mathcal{L}_i(p) \wedge \mathcal{N}_i \subseteq \bigcap_{\alpha_s \in \mathcal{L}_i(p)} \{0 \leq p_o \leq \alpha_s - 1\}. \quad (49)$$

If this property is satisfied, then the absence of heap overflows during the execution of $P$ is ensured:

**Proposition 2** *If $\mathcal{X}_N$ satisfies Property (49), then for each instruction $\langle i, *p = e,\ j\rangle$ of the program, and for any state $(i, s \mid h) \in \mathcal{C}(P)$, Property (14) is verified: dereferencing $p$ is safe.*

This proposition is the consequence of Eq. (48) and of the invariant on the well-formed memory states (Eq. (1)).

*Example 10* Let us apply the abstraction developed in this part to show the absence of heap overflows during the execution of the program $P_{receive}$ defined in Example 1. First, the sequence of the $\mathcal{X}_n$ defined by Corollary 1 is computed.

Starting from the initial state $\mathcal{X}_0 = \bot$, we have:

$$\mathcal{X}_0 = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \emptyset, t_\ell \mapsto \emptyset\}, \emptyset) \\ 2 \mapsto (\{p_\ell \mapsto \emptyset, t_\ell \mapsto \emptyset\}, \emptyset) \\ \quad \vdots \\ 12 \mapsto (\{p_\ell \mapsto \emptyset, t_\ell \mapsto \emptyset\}, \emptyset) \end{array} \right. .$$

We omit all control points whose value remains unchanged.

The state $\mathcal{X}_1$ results from the allocation of a new block at the site $\alpha$.

$$\mathcal{X}_1 = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\omega\}\}, \{i \geq 0, n \geq 0, sz \geq 0, tmp \geq 0, \alpha_s = 0\}) \\ \quad \vdots \end{array} \right. .$$

Writing a character from an external source in $sz$ lets its value unchanged: we do not have any precise information concerning this character, except its **uchar** type:

$$\mathcal{X}_2 = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\omega\}\}, \{i \geq 0, n \geq 0, sz \geq 0, tmp \geq 0, \alpha_s = 0\}) \\ 2 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. .$$

The states $\mathcal{X}_3$, $\mathcal{X}_4$, $\mathcal{X}_5$, and $\mathcal{X}_6$ correspond to the normalization of $sz$ to a value smaller than $n$ (at the control point 5), and the initializations of $i$ to 0 and $p$ to $t$ :

$$\mathcal{X}_3 = \left\{ \begin{array}{l} \quad \vdots \\ 2 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 3 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_4 = \left\{ \begin{array}{l} \quad \vdots \\ 3 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 4 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, \alpha_s = n \leq sz - 1\} \end{array} \right) \\ 5 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_5 = \left\{ \begin{array}{l} \quad \vdots \\ 4 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, \alpha_s = n \leq sz - 1\} \end{array} \right) \\ 5 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 6 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_6 = \left\{ \begin{array}{l} \quad \vdots \\ 6 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 7 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, p_o = t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. .$$

The states $\mathcal{X}_7$, $\mathcal{X}_8$, $\mathcal{X}_9$, and $\mathcal{X}_{10}$ correspond in particular to a first propagation of the over-approximations in the body of the loop located at the control point 7:

$$\mathcal{X}_7 = \left\{ \begin{array}{l} \quad \vdots \\ 7 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, p_o = t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 8 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \\ 12 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, tmp \geq 0, 0 = p_o = t_o = i = sz \leq \alpha_s = n\} \end{array} \right) \end{array} \right. ,$$

$$\mathcal{X}_8 = \left\{ \begin{array}{l} \quad \vdots \\ 8 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 9 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_9 = \left\{ \begin{array}{l} \quad \vdots \\ 9 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 10 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_{10} = \left\{ \begin{array}{l} \vdots \\ 10 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 11 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = 1, t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

In the state $\mathcal{X}_{11}$, the memory state associated to the control point 7 is modified according to the memory state of the point 11. Then, the widening operator returns a memory state in which $i = p_o \leq sz \leq \alpha_s = n$:

$$\mathcal{X}_{11} = \left\{ \begin{array}{l} \vdots \\ 7 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq i = p_o \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \\ 11 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = 1, t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

The states $\mathcal{X}_{12}$, $\mathcal{X}_{13}$, $\mathcal{X}_{14}$, and $\mathcal{X}_{15}$ correspond to a second propagation of the abstract memory state in the body of the loop:

$$\mathcal{X}_{12} = \left\{ \begin{array}{l} \vdots \\ 7 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq i = p_o \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 8 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \\ 12 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq p_o = i = sz \leq \alpha_s = n\} \end{array} \right) \end{array} \right. ,$$

$$\mathcal{X}_{13} = \left\{ \begin{array}{l} \vdots \\ 8 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 9 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. ,$$

$$\mathcal{X}_{14} = \left\{ \begin{array}{l} \vdots \\ 9 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 10 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. ,$$

$$\mathcal{X}_{15} = \left\{ \begin{array}{l} \vdots \\ 10 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 11 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o - 1 \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

The state $\mathcal{X}_{16}$ is identical to $\mathcal{X}_{15}$, since $\mathcal{F}(\mathcal{X}_{15}) \sqsubseteq \mathcal{X}_{15}$ :

$$\mathcal{X}_{16} = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\omega\}\}, \{i \geq 0, n \geq 0, sz \geq 0, tmp \geq 0, \alpha_s = 0\}) \\ 2 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 3 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 4 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n \leq sz - 1\} \end{array} \right) \\ 5 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 6 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i = 0, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 7 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 8 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 9 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 10 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 11 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o - 1 \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 12 \mapsto \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq p_o = i = sz \leq \alpha_s = n\} \end{array} \right) \end{array} \right.$$

Therefore, the limit of the sequence is reached.

Now let us check that Condition (49) is verified by $\mathcal{X}_{16}$. The only pointer dereferencing $*p = tmp$ is located at the control point 9. We have:

$$\begin{aligned} \mathcal{X}_{16}(9) &= \left( \begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ &\stackrel{def}{=} (\mathcal{L}_9, \mathcal{N}_9), \end{aligned}$$

so that $\omega \notin \mathcal{L}_9(p_\ell) = \{\alpha\}$, and $\mathcal{N}_9 \subseteq \{0 \leq p_o \leq \alpha_s - 1\}$. Proposition 2 enables us to conclude that the program $P_{receive}$ will not cause any heap overflow during its execution.

Let us remark that without the normalization of the value of $sz$ to a value smaller than $n$ (control points 3 and 4), Program $P_{receive}$ could lead to a heap overflow, and the analysis would have correctly raised an alarm: the numerical invariant at the control point 9 would have been $\mathcal{N}_9 = \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, \alpha_s = n \geq 1\}$. But, $\mathcal{N}_9 \not\subseteq \{0 \leq p_o \leq \alpha_s - 1\}$.

Moreover, if the domain of intervals would have been used instead of polyhedra (as in [31]), the numerical invariant at 9 would have been:

$$\begin{aligned} \mathcal{R}_9 = \{&t_0 \mapsto [0; 0], tmp \mapsto [0; +\infty[, \\ &p_o \mapsto [0; +\infty[, i \mapsto [0; +\infty[, \\ &n \mapsto [1; +\infty[, \alpha_s \mapsto [1; +\infty[, sz \mapsto [1; +\infty[\}, \end{aligned}$$

which is not accurate enough to ensure that $0 \leq p_o \leq \alpha_s - 1$. Consequently, a more accurate domain, such as convex polyhedra, is required to show the absence of defects in $P_{receive}$.

## 5 Some possible extensions

The formalism presented in the previous sections can be enriched so as to cover most of the features of programming languages such as C language. In this section, we present some examples of possible extensions.

First, more complex conditions can be easily introduced in the kernel language, by adding the conjunction and the disjunction of conditions (logical clauses). Their semantics are defined similarly to the rules given in Fig. 8, and their abstraction are performed by using union and intersection operators.

Moreover, the instruction $x = *p$ which consists in reading a data contained in the heap and then assigning it in a stack variable can be additionally considered. The corresponding rule in the concrete semantics is defined by:

$$\frac{p = (l_\alpha, o) \in \mathsf{dom}(h) \quad h(l_\alpha, o) = v}{s \,\natural\, h \vdash x = *p : s[x \mapsto v] \,\natural\, h}, \tag{50}$$

and its abstract counterpart can be defined by:

$$\begin{aligned}&[\![x = *p]\!](\mathcal{L}, \mathcal{N})\\ &\quad = (\mathcal{L}[p_\ell \mapsto \mathcal{L}(p_\ell) \cap \mathsf{Alloc}], (|x \geq 0|) \circ (|x \leftarrow ?|)(\mathcal{N}))\,,\end{aligned} \tag{51}$$

which assigns a random value to $x$ (since the abstraction of the heap does not provide any information on the value of $*p$). Besides, the validity of the dereferencing of $p$ when reading the value pointed to, can be added to the properties to be verified.

Several other features can be added to the language: their formalization raises independent issues to those encountered here: (i) other integer types, such as Booleans, signed or unsigned integers, enumerations, various qualifiers (for instance **short** and **long** in C language), and the analysis of integer overflows in computations [3], (ii) the **null** pointer, pointers with arbitrary depth, pointers to the stack, multiple dereferencing, (multidimensional) arrays, structures, unions, string buffers [36,1], (iii) a richer control flow, with calls to non-recursive functions, jumps, several kinds of loops, local variable declarations (see for example [1]).

Finally, the computationally complex domain of polyhedra can be replaced by a less accurate but more efficient one. For instance, octagons [35] (numerical invariants of the form $\pm x \pm y \leq c$, where $x$ and $y$ are variables, $c$ a constant) would have been sufficient to analyze the program $P_{receive}$.

## 6 Related work

Most static analyzers are not *sound* and only point out some potential bugs to the user. That is, when all the errors they detect are corrected, it is still not sure that the program is safe. In practice, some reachable machine states are forgotten during the analysis (often to improve the time or memory performances of the analyzer). Consequently, they cannot be used to ensure the security of critical software. Several methods are used in these analyzers: (i) pattern matching (GNU grep [27]), (ii) heuristics (flawfinder [22], ITS4 [45]), (iii) and more generally, propagation of properties (numerical properties, pointer aliasing), but the propagation is not sound (Splint [19], Coverity [14], BOON [46], EauClaire [7], CCA [6], Uno [28]). In particular, these tools handle the detection of uninitialized variables, **null** pointer dereferencing, illegal array accesses, and some other library functions which may be dangerous (such as strcpy in C language). The analyzed languages are, for instance, C and C++, Java, and ADA.

Other static analyzers are based on *sound* formal methods. Besides abstract interpretation, we can mention: (i) model checking [8,18], whose principle is to exhaustively explore the set of reachable states of the system (as this can be achieved only if the analyzed model is indeed finite, the exploration can only be performed on finite abstractions of the model [9]), (ii) predicate abstraction [26], a model checking technique that uses Boolean model generated from predicates initially defined. Then other predicates can be added to refine the initial model if necessary [2,15], (iii) theorem proving techniques [20,21], which translates the program semantics and the properties to be verified into logic formulas, and then tries to (semi)-automatically prove them with a proof assistant.

The analyses based on abstract interpretation handle several properties of programs. For example, (i) numerical properties on integers (integer overflows [3,38]) and on floating-point numbers (undefined operations [3], numerical precision [25]), (ii) numerical properties on pointers (stack overflows [31,36,44]) and the length of string buffers [1,17], (iii) properties on the shape of the memory (lists and trees [41], partitioning into disjoint parts [16]), (iv) invariants on classes in object-oriented languages [33], (v) security properties of cryptographic protocols [4,5].

## 7 Conclusion

We have built a static analysis based on abstract interpretation which allows to show the absence of heap overflows in a program. In particular, the program given in the introduction has been successfully analyzed. Moreover, several extensions have been discussed.

Static analysis by *sound* formal methods allows to ensure that a software does not contain any security flaw. This is fundamental for the implementation of reliable critical software infrastructures.

**Acknowledgments** Thanks to Alexandre Gazet, Wenceslas Godard, and Eric Goubault for their useful comments.

## References


1. Allamigeon, X., Godard, W., Hymans, C.: Static analysis of string manipulations in critical embedded C programs. In: Yi, K. (ed.) Static Analysis, 13th International Symposium (SAS'06), Volume 4134 of Lecture Notes in Computer Science, pp. 35–51, Seoul, Korea, August 2006. Springer, Heidelberg (2006)
2. Ball, T., Cook, B., Das, S., Rajamani, S.: Refining approximations in software predicate abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 388–403. Springer, Heidelberg, March 2004
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: ACM SIGPLAN PLDI'03, Volume 548030, pp. 196–207. ACM, New York, June 2003
4. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14), pp. 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society (2001)
5. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: IEEE Symposium on Security and Privacy, pp. 140–154, Oakland, CA, May 2006
6. C Code Analyzer. http://www.drugphish.ch/~jonny/cca.html
7. Chess, B.: Improving computer security using extended static checking. In: SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp. 160, Washington, DC, USA, 2002. IEEE Computer Society (2002)
8. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
9. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252, Los Angeles, CA. ACM, New York (1977)
11. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pac. J. Math. **82**(1), 43–57 (1979)
12. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4), 511–547 (1992)
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 84–97, Tucson, Arizona, 1978. ACM, New York
14. Coverity. http://www.coverity.com
15. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: Formal Methods in Computer-Aided Design. Springer, Heidelberg, November 2002
16. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006, Volume 3920 of Lecture Notes in Computer Science, pp. 287–302. Springer, Heidelberg, March 2006
17. Dor, N., Rodeh, M., Sagiv, M.: Cssv: towards a realistic tool for statically detecting all buffer overflows in C. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 155–167, New York, NY, USA. ACM, New York (2003)
18. Jr. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT, Cambridge (1999)
19. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Softw. **19**(1), 42–51 (2002)
20. Filliâtre, J.-C.: Verification of non-functional programs using interpretations in type theory. J. Funct. Program. **13**(4), 709–745 (2003)
21. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004. Volume 3308 of Lecture Notes in Computer Science, pp. 15–29. Springer, Heideleberg (2004)
22. Flawfinder. http://www.dwheeler.com/flawfinder/
23. International Organization for Standardization. ISO/IEC 9899:1999: Programming Languages—C. International Organization for Standardization, Geneva, Switzerland, December 1999
24. Ganssle, J.: Big Code. http://www.embedded.com/columns/embeddedpulse/171203287?_requestid=1130518
25. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) Static Analysis, 13th International Symposium (SAS'06), Volume 4134 of Lecture Notes in Computer Science, pp. 18–5134, Seoul, Korea, August 2006. Springer Verlag (2006)
26. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97), Vol. 1254, pp. 72–83. Springer Verlag (1997)
27. GNU grep. http://www.gnu.org/software/grep/
28. Holzmann, G.J.: Static source code checking for user-defined properties. In: Proceedings IDPT 2002, Pasadena, CA, USA (2002)
29. Hymans, C., Levillain, O.: Newspeak: Big Brother is compiling your code. Technical report, EADS France (2007). http://www.penjili.org/newspeak.html
30. Ghidella, J.R., Friedman, J.: Streamlined development of body electronics systems using model-based design. http://www.mathworks.com/company/pressroom/newsletter/sept06/body_electronics.html
31. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In: Siveroni, I., Hankin, C. (eds.) Static Analysis: 12th International Symposium, SAS 2005, London, UK, September 7–9, 2005. Proceedings, Lecture Notes in Computer Science, pp. 203–217. Springer Verlag (2005)
32. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)
33. Logozzo, F.: Automatic inference of class invariants. In: Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04), Volume 2937 of Lectures Notes in Computer Science, January 2004. Springer Verlag (2004)
34. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: PADO II, Volume 2053 of LNCS, pp. 155–172, May 2001. Springer Verlag. http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf
35. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001, IEEE, pp. 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf
36. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: ACM SIGPLAN LCTES'06, pp. 54–63. ACM, New York, June 2006. http://www.di.ens.fr/~mine/publi/article-mine-lctes06.pdf
37. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, pp. 213–228, London, UK. Springer Verlag (2002)

38. Polyspace. http://www.polyspace.com
39. Rice, H.G.: On completely recursively enumerable classes and their key arrays. J. Symb. Log. **21**(3), 304–308 (1956)
40. Pehrson, R.J.: Software development for the Boeing 777. http://www.stsc.hill.af.mil/crosstalk/1996/01/Boein777.asp
41. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Symposium on Principles of Programming Languages, pp. 105–118 (1999)
42. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) Verification, Model Checking and Abstract Interpretation: Proceedings of the 6th International Conference (VMCAI 2005), Volume 3385 of Lecture Notes in Computer Science, pp. 25–41, Paris, France, 2005. Springer, Berlin (2005)
43. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**, 285–309 (1955)
44. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pp. 231–242, New York, NY, USA. ACM, New York (2004)
45. Viega, J., Bloch, J.T., Kohno, Y., McGraw, G.: Its4: A static vulnerability scanner for C and C++ code. In: ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference, pp. 257, Washington, DC, USA. IEEE Computer Society (2000)
46. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Network and Distributed System Security Symposium, pp. 3–17, San Diego, CA, February 2000
47. Wikipedia. Source lines of code—Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Lines_of_code