Abstract Arithmetisation of the Heap

Frédéric Besson

Inria Rennes - Bretagne Atlantique

Abstract We propose a non-standard approach for analysing heap-manipulating programs using a purely numeric abstract semantics which encodes alias information using only 7 additional *heap variables*. The abstraction is expressive enough to capture properties pertaining to (recursive) array contents and numeric relationships between both pointers and integer variables. The numeric abstraction is interpreted over a time-stamp program semantics identifying a pointer by a triple of integers. The abstract semantics is expressed using standard set operations and can be further abstracted by numeric abstract domains. Experiments show that the abstract semantics is expressive and captures non-trivial invariants of heap-manipulating programs *e.g.,* sorting, matrix and list algorithms. Moreover, the abstract semantics ensures by construction that abstract verification conditions are decidable as soon as the abstract invariants and all the assignments in the program are expressed in the linear fragment of arithmetic.

1 Introduction

The prominent method for the precise analysis of heap-manipulating programs is to combine a numeric abstract domain [7,31,27,15,5] with an abstract memory model [9,33,30,26,14,20,24,29] keeping track of the shapes in the heap. In this paper, we investigate an alternative approach which consists in devising a purely numeric abstract semantics which is expressive enough to capture invariants about programs manipulating both recursive linked data-structures and the content of arrays.

Previous works (see *e.g.*, [34,11,22,23]) have investigated how to reduce programs with pointers to numeric programs. Although the techniques are different, the aforementioned work have in common a two-staged approach: a numeric program is obtained by running a preliminary pointer (or shape) analysis which explicits alias relations. This contrasts with our proposed abstract semantics which operates directly over the heap-manipulating program without requiring a prior analysis. A direct numeric abstraction of heap-manipulating programs is due to Bouajjani *et al.* [4]. They show that programs manipulating singly-linked lists only are equivalent to counter automata. Our abstract semantics is not limited to singly-linked lists but, at the expense of approximation, can analyse both more complex heap data-structures.

An important property of our abstract semantics is that it avoids Minsky's tar-pit. It is almost certain that analysing Minsky machines obtained from arbi-

trary heap-manipulating programs is doomed or would require incredibly powerful abstract numeric domains. Our abstract semantics operates directly over the semantics of heap-manipulating programs; and linear arithmetic is sufficient to express abstract invariants of typical heap-manipulating programs.

Our objective is to design a numeric abstract semantics for heap-manipulating programs. To our knowledge, for general heap-manipulating programs, this extreme approach is original and has never been attempted before. In essence, we show how to abstract a set of program states $s \subseteq State$ by a set of (finite) vectors $s^\sharp \subseteq \mathbb{N}^n$ where n is given by

$$n = i + 3 * p + r + 7 * k$$

where i is the number of integer variables, p is the number of pointer variables, r is the number of syntactic memory regions and k is a precision parameter. In our semantics, a pointer is a triple of integers thus explaining the $3 * p$. The encoding of the heap requires 7 additional indexed heap variables.

$$HeapVar_i \triangleq \{reg_o^i, ts_o^i, of_o^i, size^i, reg_d^i, ts_d^i, of_d^i\}$$

The subscript o stands for *origin* and the subscript d stands for *destination*. Intuitively, (reg_o^i, ts_o^i) represents the base address of an arbitrary memory block, say mb, in the domain of the heap; $size^i$ is the size of the memory block mb and the pointer $((reg_d^i, ts_d^i), of_d^i)$ represents the content of the memory block mb at offset of_o^i. The parameter k specifies the number of instances of heap variables to be used by the abstraction. It is not correlated to the standard *k-limiting* approach [19] to alias analysis and all our examples, which consider data-structures of arbitrary size, can be analysed using values of k no more than 2.

Our abstract semantics has the following distinctive features:

- the abstraction is expressive and captures sophisticated invariants of recursive linked data-structures and array contents;
- a decidable fragment of arithmetic, namely integer linear arithmetic, is sufficient to express the invariants and discharge abstract verification conditions of typical programs;
- abstract transformers can be readily computed by off-the-shelf numeric abstract domains.

The inference of abstract invariants is not in the scope of the present paper. For our program examples, the abstract invariants are linear and could therefore in theory be inferred by existing numeric abstract domains such as the disjunctive completion [6] of convex polyhedra [7] or Presburger automata [36].

From a methodological standpoint, we argue that a purely numeric abstract semantics has the virtue of simplifying the design of derived static analyses. In particular, it provides an alternative for sophisticated (and largely non-automatic) collaboration schemes such as reduced products [6] or co-fibered domains [33]. A purely numeric abstract semantics offers *for free* an optimal collaboration between numeric variables and (numerically encoded) alias information.

The organisation of the rest of the paper is as follows. In Section 2 we devise a time-stamp semantics for imperative heap-manipulating programs. The semantics is fairly standard: a pointer is represented by an offset w.r.t. the base address of a memory block which is an array of pointers. Section 3 presents the rationale for the abstract domain and illustrates its expressiveness with examples. A key feature of the abstraction is that time-stamps are not abstracted and therefore an abstract pointer is nothing but a concrete pointer. This is instrumental for the abstract semantics to model strong-updates in the heap. The content of the heap is modelled using 7 additional numeric variables allowing to describe a relation between a virtual pointer p and its de-reference $*p$. Section 4 details the abstract semantics and shows that it can be expressed using standard set operators. Section 5 is devoted to our mechanised formalisation and experiments. The abstract semantics has been mechanically proved correct with respect to the concrete semantics [2] using the Coq proof-assistant. Abstract verification conditions used to validate abstract invariants are generated using code automatically extracted from the Coq development. We show that the invariants of typical programs manipulating arrays, lists and even trees can be expressed using our numeric abstract semantics. Furthermore, the invariants are expressed in a decidable fragments of arithmetic, namely linear integer arithmetic, and are discharged automatically by off-the-shelf automatic provers. Section 6 is devoted to related work and Section 7 concludes.

2 Time-stamp semantics

This section presents the syntax and semantics of our kernel language. The language is idealised and therefore i) the only primitive types are integers and pointers; ii) the heap is a total function mapping addresses to arrays of pointers; iii) the semantics is non-blocking and error-free. Nonetheless, the language captures essential features of more realistic languages. In particular, our kernel language can express traditional safety policies such as memory safety, the absence of double-free or the absence of array-out-of-bound accesses. These properties or more sophisticated invariants can be ensured (and statically checked by the abstract semantics) by inserting dynamic assertions. Subsection 2.1 explains the rationale for our memory model. Subsection 2.2 describes the syntax and operational semantics of the language.

2.1 Memory model

Operational semantics usually consider an abstract infinite uninterpreted domain *Addr* of addresses. Memory allocation is therefore modelled by picking an arbitrary *fresh* address. In order to track addresses numerically a more concrete numeric representation is needed. A canonical choice is to identify the set of addresses *Addr* with the set $\mathbb{N}$ of natural numbers and increment a counter at each allocation. This is that very scheme that is implemented by the COMPCERT verified compiler [21]. Such a model has the advantage of simplicity but

$$
\begin{array}{ll}
Region \triangleq [1,\ldots,k] & Addr \triangleq Region \times \mathbb{N} \\
Pointer \triangleq Addr \times \mathbb{N} & Array \triangleq \mathbb{N} \times (\mathbb{N} \to Pointer) \\
Heaps \triangleq Addr \to Array & \\
Store \triangleq (V_\iota \to \mathbb{N}) \times (V_\pi \to Pointer) \times (Region \to \mathbb{N}) & \\
State \triangleq Label \times Store \times Heap &
\end{array}
$$

Figure 1. Semantic domains

would put some unnecessary stress on numeric abstract domains to identify uncorrelated ranges of addresses. For our purpose, a better choice consists in partitioning *a priori* the set of addresses into regions. Hence, an address is a pair $(r, ts) \in Region \times \mathbb{N}$ where r ranges over a finite number of *regions* and ts is the time-stamp local to region r. The regions are statically defined and this might seem overly restrictive. Section 4.2 discusses how to accommodate for more dynamic partitioning of the heap.

Given our concrete notion of address, our semantic domains are similar to existing models for low-level code *e.g.*, [14]. The heap maps an address to a memory block which has a given size and is an array of pointers. A pointer is made of a base address to a memory block paired with an integer offset. A consequence is that an integer i is cast into the pointer $((0,0), i)$ before being written to the heap. Nonetheless, pointers and integers do not mix up as allocated addresses have by construction an immutable strictly positive time-stamp. The `null` pointer is modelled by $((0,0),0)$. The collision with the encoding of the integer 0 is benign as the `null` pointer should never be de-referenced. The structure of the heap is independent of the precise memory layout and memory blocks do not overlap. For the sake of simplicity, the heap is a total function and unallocated addresses are mapped to a memory block of size 0.

The semantics state is a triple $(l, s, h) \in Label \times Store \times Heap$ where $l \in Label$ is a program label, $s \in Store$ is an environment mapping integer variables to integers, pointer variables to pointer values and mapping regions to their time-stamp. Semantics domains are summarised in Figure 1 where V_ι is the set of integer variables and V_π is the set of pointer variables.

2.2 Program syntax and semantics

A program P is a list of commands labelled by integer labels. Labels are unique and are used to indicate the target of branching and jumping commands. Sequencing is implicit and consists in incrementing $(+1)$ the current label. The syntax of programs is given in Figure 2. Most commands are standard and correspond to instructions found in classic languages. Integer expressions allow to completely introspect the semantic state. For instance, $x^\pi.$`len` returns the length of the array in the heap at address x^π. The region, time-stamp and offset of a pointer x^π can obtained by the expressions $x^\pi.$`reg`, $x^\pi.$`ts` and $x^\pi.$`of` respectively. The expression `r` returns the current time-stamp of the region r. The expression

$$
\begin{array}{lll}
Variables & x^\tau & \in\ V_\tau \quad \tau \in \{\iota, \pi\} \\
Numbers & n & \in\ \mathbb{N} \\
Operators & \bowtie & \in\ \{+, -, *\} \\
Integer\ Exprs & e^\iota & ::= ?\ |\ n\ |\ x^\iota\ |\ x_1^\iota \bowtie x_2^\iota\ |\ x^\pi.\texttt{len}\ |\ x^\pi.\texttt{reg}\ |\ x^\pi.\texttt{ts}\ |\ x^\pi.\texttt{of}\ |\ \texttt{r} \\
Pointer\ Exprs & e^\pi & ::= \texttt{null}\ |\ x^\pi\ |\ x^\pi + x^\iota\ |\ *x^\pi\ |\ x^\iota \\
Commands & c & ::= x^\iota := e^\iota\ |\ x^\pi := e^\pi\ |\ *x_1^\pi := x_2^\pi\ |\ x^\pi := \texttt{new}[x^\iota]^{\texttt{r}}\ |\ \texttt{free}(x^\pi) \\
& & \quad |\ \texttt{assumez}(x^\iota)\ |\ \texttt{ensurez}(x^\iota)\ |\ \texttt{goto}\ l'\ |\ \texttt{jz}(x^\iota) \Rightarrow l'
\end{array}
$$

Figure 2. Program syntax

? returns an arbitrary integer and is the only non-deterministic expression. The
`null` pointer is coded by $((0,0),0)$. The expression $x^\pi + x^\iota$ performs pointer
arithmetic and increments the offset of the pointer x^π by the value of x^ι. The
expression $*x^\pi$ de-references the pointer x^π. The pointer expression x^ι casts an
integer into a pointer: the integer i is represented by $((0,0),i)$. As the address
$(0,0)$ can never be obtained by an allocation instruction (time-stamps are incre-
mented before the allocation) a cast integer cannot be confused with a genuine
pointer. This coding allows however to store integers in the heap without com-
plicating the semantic domain.

The semantics of integer and pointer expressions is given in Figure 3. The
rules use the following notations. Given a pointer p, we write $p.reg$, $p.ts$ and
$p.of$ for its region, time-stamp and offset, respectively. As a result, we have
$p = ((p.reg, p.ts), p.of)$. We also write $p.\alpha$ for the base address of the pointer
p i.e., $(p.reg, p.ts)$. We write $f[x \mapsto v]$ for the function that is the same as f
except that x is mapped to v. We use this notation for updating environments,
heaps and arrays. An array of size n containing only `null` pointers is written 0_n.
Given an array a, $a.len$ returns the length of the array and $a[i]$ accesses the ith
element of the array.

$$
\begin{aligned}
&\llbracket ? \rrbracket(s,h) = \mathbb{N} \\
&\llbracket n \rrbracket(s,h) = \{n\} \\
&\llbracket x^\iota \rrbracket(s,h) = \{s(x^\iota)\} \\
&\llbracket x_1^\iota \bowtie x_2^\iota \rrbracket(s,h) = \{s(x_1^\iota) \bowtie s(x_2^\iota)\} \\
&\llbracket x^\pi.\texttt{reg} \rrbracket(s,h) = \{s(x^\pi).reg\} \\
&\llbracket x^\pi.\texttt{ts} \rrbracket(s,h) = \{s(x^\pi).ts\} \\
&\llbracket x^\pi.\texttt{of} \rrbracket(s,h) = \{s(x^\pi).of\} \\
&\llbracket x^\pi.\texttt{len} \rrbracket(s,h) = \{h(s(x^\pi).\alpha).len\} \\
&\llbracket \texttt{r} \rrbracket(s,h) = \{s(r)\}
\end{aligned}
\qquad
\begin{aligned}
&\llbracket \texttt{null} \rrbracket(s,h) = ((0,0),0) \\
&\llbracket x^\pi \rrbracket(s,h) = s(x^\pi) \\
&\llbracket x^\pi + x^\iota \rrbracket(s,h) = (s(x^\pi).\alpha, s(x^\pi).of + s(x^\iota)) \\
&\llbracket x^\iota \rrbracket(s,h) = ((0,0), s(x^\iota)) \\
&\llbracket *x^\pi \rrbracket(s,h) = h(s(x^\pi).\alpha)[s(x^\pi).of]
\end{aligned}
$$

$\qquad\qquad$ (a) Integer expressions $\qquad\qquad\qquad$ (b) Pointer expressions

Figure 3. Semantics of expressions

$$\dfrac{x^\iota{:=}e^\iota \quad v \in [\![e^\iota]\!](s,h)}{(l,s,h) \to (l{+}1, s[x^\iota \mapsto v], h)} \qquad \dfrac{x^\pi{:=}e^\pi \quad v = [\![e^\pi]\!](s,h)}{(l,s,h) \to (l{+}1, s[x^\pi \mapsto v], h)} \qquad \dfrac{\texttt{goto } l'}{(l,s,h) \to (l', s, h)}$$

$$\dfrac{\texttt{free}(x^\pi)}{(l,s,h) \to (l{+}1, s, h[s(x^\pi).\alpha \mapsto 0_0])} \qquad \dfrac{*x_1^\pi{:=}x_2^\pi \quad s(x_1^\pi) = (\alpha, o)}{(l,s,h) \to (l{+}1, s, h[\alpha \mapsto h(\alpha)[o \mapsto s(x_2^\pi)]])}$$

$$\dfrac{x^\pi{:=}\texttt{new}[x^i]^{\texttt{r}} \quad vr = s(r) + 1 \quad \alpha = (\texttt{r}, vr)}{(l,s,h) \to (l{+}1, s[r \mapsto vr][x^\pi \mapsto (\alpha, 0)], h[\alpha \mapsto 0_{s(x^i)}])} \qquad \dfrac{\texttt{jz}(x^\iota) \Rightarrow l' \quad s(x^\iota) = 0}{(l,s,h) \to (l', s, h)}$$

$$\dfrac{\texttt{check}(x^\iota) \quad \texttt{check} \in \{\texttt{assumez}, \texttt{ensurez}\} \quad i = 1 - s(x^\iota)}{(l,s,h) \to (l{+}i, s, h)} \qquad \dfrac{\texttt{jz}(x^\iota) \Rightarrow l' \quad s(x^\iota) \neq 0}{(l,s,h) \to (l{+}1, s, h)}$$

Figure 4. Operational semantics

The semantics of statements is given in Figure 4 and takes the form of a transition relation $\cdot \to \cdot \subseteq State \times State$ between states. In the rules, the current command is implicitly labelled by the label l. The assignment $x^\iota{:=}e^\iota$ (resp., $x^\pi{:=}e^\pi$) updates integer variables (resp. pointer variables). The command $*x_1^\pi := x_2^\pi$ writes the value of the pointer x_2^π at the address pointed by x_1^π. For simplicity, we assume that the variable x_1^π is distinct from x_2^π. This is a syntactic restriction that can be circumvented by introducing an auxiliary variable. Allocation of an array of size x^ι in a region $\texttt{r}$ is done by the statement $x^\pi{:=}\texttt{new}[x^\iota]^{\texttt{r}}$. The statement $\texttt{free}(x^\pi)$ de-allocates the array referenced by the pointer x^π. The statements $\texttt{assumez}(x^\iota)$ and $\texttt{ensurez}(x^\iota)$ have the same dynamic semantics: if the value of variable x^ι is zero the execution continues sequentially; otherwise the execution is stuttering on the current label. However, from a verification standpoint the interpretation is different: a program is valid if for all the $\texttt{ensurez}$ instructions the value of the variable x^ι is provably 0. The $\texttt{assumez}$ statement is used to express pre-conditions. Control-flow is altered by the statements $\texttt{goto } l'$ and $\texttt{jz}(x^\iota) \Rightarrow l'$. The former performs an unconditional jump to l' while the latter only performs a jump to l' when the value of the integer x^ι is zero.

3   Abstract domain

This section presents our abstraction of the heap. In Subsection 3.1, we define the abstract domain and its concretisation function. Subsection 3.2 illustrates the expressiveness of the abstraction with examples.

3.1   Numeric heap abstraction

As stated in the introduction, we will abstract a set of program states $s \subseteq Store \times Heap$ by a set of (finite) vectors $s^\sharp \subseteq \mathbb{N}^{|n|}$. A direct consequence of our time-stamps semantics is that a store $s \in Store$ is isomorphic to a numeric vector:

$$Store \simeq \mathbb{N}^{|V_\iota| + 3 \times |V_\pi| + |Region|}$$

Unsurprisingly, the semantic state of programs manipulating integers and pointers (but not accessing the heap) can be modelled numerically without any kind of abstraction. In our model, a heap $h \in Heap$ is a total function of type $\mathbb{N}^2 \to (\mathbb{N} \times (\mathbb{N} \to \mathbb{N}^3))$ (see Figure 1). This functional space is not isomorphic to any finite product domain $\mathbb{N} \times \cdots \times \mathbb{N}$ and abstraction is therefore mandatory. An objection to this combinatorial argument is that a heap can actually be encoded by a single integer because programs can only allocate a finite amount of memory. This objection is theoretically valid but does not lead to a tractable numeric encoding. On the contrary, our abstraction is relatively precise while allowing to capture sophisticated invariants using only the expressive power of linear arithmetic.

Jeannet *et al.,* [17] have investigating several ways of abstracting sets of functions $F \subseteq D_1 \to D_2$. They make the pervasive assumption that $\mathcal{P}(D_1)$ is abstracted by a domain A_1 of finite cardinality. We follow a different approach that makes this assumption irrelevant. Our starting point is the classic abstraction of functional spaces [17, Section 3.1] but with two tweaks that are instrumental for improving precision.

Definition 1 (Classic abstraction of functional spaces). *A set of functions $F \subseteq D_1 \to D_2$ is abstracted by a relation $f^\sharp \subseteq D_1 \times D_2$. A function f belongs to the concretisation of $f^\sharp$ if, seen as a relation, it is contained in $f^\sharp$.*

$$f \in \gamma(f^\sharp) \text{ iff } \forall x, (x, f(x)) \in f^\sharp$$

To adapt this construction to the heap domain, we introduce 7 heap variables

$$HeapVar \triangleq \{reg_o, ts_o, of_o, size, reg_d, ts_d, of_d\}$$

and name *heap-slice* the function space $HeapVar \to \mathbb{N}$ mapping heap variables to integers.

$$HeapSlice \triangleq HeapVar \to \mathbb{N}$$

A heap-slice expresses a relation between pointers in a heap h and has the following intuitive meaning:

- (reg_o, ts_o) is an arbitrary address (the index o stands for origin);
- $size$ is the size of the array at address (reg_o, ts_o) ($h(reg_o, ts_o).len = size$);
- $((reg_d, ts_d), of_d)$ is the pointer obtained by de-referencing in the heap the pointer $((reg_o, ts_o), of_o)$ (the index d stands for destination).

Given a heap h and a heap-slice hs, Definition 2 introduces a compatibility relation (written $hs \models h$).

Definition 2. *A heap-slice $hs \in HeapVar \to \mathbb{N}$ is compatible with a heap h ($hs \models h$) if and only if the array $h(hs(reg_o), hs(ts_o))$ has size $hs(size)$ and contains at index $hs(of_o)$ the pointer $((hs(reg_d), hs(ts_d)), hs(of_d))$. Formally, we have:*

$$hs \models h \text{ iff } \begin{array}{l} let\ a := h(hs(reg_o), hs(ts_o)) \\ in\ \bigwedge \left(\begin{array}{l} a.len = hs(size) \\ a[hs(of_o)] = ((hs(reg_d), hs(ts_d)), hs(of_d)) \end{array} \right) \end{array}$$

The *classic* abstraction consists in abstracting $\mathcal{P}(\textit{Heap})$ by $\mathcal{P}(\textit{HeapSlice})$ and the concretisation function of Definition 1 can be rewritten as

$$h \in \gamma(h^\sharp) \text{ iff } \forall hs, (hs \models h) \Rightarrow hs \in h^\sharp$$

Our construction refines this abstraction in two ways. It preserves more relational information within the heap by keeping k copies of *HeapSlice*. It also incorporates the store into the abstraction in a relational way. Formally, we obtain the parametrised abstract domain:

$$\mathbb{D}_k^\natural \triangleq \mathcal{P}(\textit{Store} \times \textit{HeapSlice}^k)$$

This domain is purely numeric and can be abstracted by existing abstract numeric domains. The concretisation function $\gamma_k : \mathbb{D}_k^\natural \to \mathcal{P}(\textit{Store} \times \textit{Heap})$ is formally defined by:

$$(s, h) \in \gamma_k(d) \text{ iff } \forall hs, (\forall i \in 1 \ldots k, hs_i \models h) \Rightarrow (s, (hs_1, \ldots, hs_k)) \in d$$

A concrete state (s, h) belongs to the concretisation of an abstract state $d^\sharp$ if for all heap-slices $(hs_1, \ldots, hs_k)$ constituting the heap h the pair $(s, (hs_1, \ldots, hs_k))$ belongs to the abstraction d. Dually, the idea behind the abstraction is to view a heap h as a set of heap-slices. A state (s, h) is abstracted by taking the product of the singleton store s with k copies of the heap-slices constituting the heap h: $\{s\} \times \{hs \mid hs \models h\}^k$. An important property of the abstraction is that it keeps a relation between the store and the heap. Another important property is that *strong-updates* are modelled by updating heap-slices. In the rest, instead of $\mathbb{D}_k^\natural$, we use the isomorphic domain $\mathbb{D}_k^\sharp$ defined by

$$V_k \triangleq V_\iota + V_\pi.\texttt{reg} + V_\pi.\texttt{ts} + V_\pi.\texttt{of} + \textit{Region} + \textit{HeapVar}_1 + \ldots + \textit{HeapVar}_k$$
$$\mathbb{D}_k^\sharp \triangleq \mathcal{P}(V_k \to \mathbb{N})$$

It is based on the observation that $A + B \to \mathbb{N} \simeq (A \to \mathbb{N}) \times (B \to \mathbb{N})$. Another transformation is that a pointer variable x^π is split into three integer variables $x^\pi.\texttt{reg}$, $x^\pi.\texttt{ts}$ and $x^\pi.\texttt{of}$ which represent respectively the region, the time-stamp and the offset of the pointer x^π. Moreover, heap variables are indexed:

$$\textit{HeapVar}_i \triangleq \{reg_o^i, ts_o^i, of_o^i, size^i, reg_d^i, ts_d^i, of_d^i\}$$

3.2 Expressiveness

The following examples illustrate the expressive power of the abstraction and demonstrate that it is sufficiently precise to capture (numerically) rather sophisticated properties of the heap. It is worth noticing that the invariants can be expressed by using linear arithmetic only and that they can all be established by the abstract transformers given in Section 4. For the sake of readability, the examples are not written in the core language but using a structured WHILE-like

pseudo-code. We also make the simplifying assumption that the time-stamp of regions ρ, ρ' is initially 0.

The following pseudo-code builds a triangular matrix of size n. It shows that the abstraction is able to precisely reason about the content of arrays.

```
m:=new[n]ρ; for(i:=0;i<n;i++) m[i]:=new[i]ρ'
```

Assuming that the time-stamp of region ρ' is initially 0, the following abstract invariant captures the fact that the matrix is indeed triangular:

$$reg_o = m.\mathbf{reg} \wedge ts_o = m.\mathbf{ts} \wedge of_o \leq n \Rightarrow reg_d = \rho' \wedge ts_d = 1 + of_o$$
$$\wedge$$
$$reg_o = \rho' \wedge 0 < ts_o \leq n \Rightarrow size = ts_o - 1$$

A consequence of our model is that linked data-structures are nothing but recursive arrays. The following pseudo-code constructs a linked-list of size n.

```
hd:=new[2]ρ; lst := hd;
for(i:=0;i<n;i++)
{lst[0]:=?; rst:=new[2]ρ; lst[1]:=rst; lst:=rst}
```

The cell node `lst` is an array of size 2 ; `lst[0]` is the element of the list and `lst[1]` is the *next* field. In this case, the abstraction precisely captures the fact that the heap contains indeed a linked-list of length n.

$$reg_o = \rho \wedge 0 < ts_o \leq n \wedge of_o = 1 \Rightarrow reg_d = \rho \wedge ts_d = ts_o + 1 \wedge of_d = 0$$
$$\wedge$$
$$reg_o = \rho \wedge ts_o = n + 1 \Rightarrow reg_d = 0 \wedge ts_d = 0 \wedge of_d = 0$$

Providing an adequate allocation pattern, tree-like structures can also be precisely modelled by the abstraction. The following pseudo-code constructs trees from an array of nodes.

```
sz := 2*n+1; t = new[sz]ρ;
for(i:=0; i<sz ; i++){ nd := new[2]ρ'; t[i] := nd };
for (i:=0; i<n ; i++){
  if(?) t[i][0] := t[2*i+1] else t[i][0] := null ;
  if(?) t[i][1] := t[2*i+2] else t[i][1] := null ;
};
```

The abstraction precisely captures the tree structure and would be therefore be sufficient to ensure the absence of dangling pointers.

$$reg_o = \rho' \wedge 0 < ts_o \leq n \wedge of_o = 0 \Rightarrow \bigvee \left(\begin{array}{c} reg_d = \rho' \wedge ts_d = 0 \wedge of_d = 0 \\ reg_d = \rho' \wedge ts_d = 2 * ts_o \wedge of_d = 0 \end{array} \right.$$
$$reg_o = \rho' \wedge 0 < ts_o \leq n \wedge of_o = 1 \Rightarrow \bigvee \left(\begin{array}{c} reg_d = \rho' \wedge ts_d = 0 \wedge of_d = 0 \\ reg_d = \rho' \wedge ts_d = 2 * ts_o + 1 \wedge of_d = 0 \end{array} \right.$$
$$reg_o = \rho' \wedge 1 + n \leq ts_o \leq 2 * n + 1 \Rightarrow reg_d = 0 \wedge ts_d = 0 \wedge of_d = 0$$

The previous invariants are expressed using only a single heap-slice *i.e.*, the domain $\mathbb{D}_1^\sharp$. This abstraction is not powerful enough to capture for instance the

property *being sorted*. The reason is that $\mathbb{D}_1^\sharp$ can only indirectly express relations between different heap-slices by mean of other numeric variables. The domain $\mathbb{D}_k^\sharp$ for some k $(k > 1)$ lifts this restriction and allows to express relations between k distinct locations in the heap. As a result, the domain $\mathbb{D}_2^\sharp$ can express the invariant of sorting algorithms. For instance, the invariant

$$\forall i, j, n, i \leq j \leq n \Rightarrow t[i] \leq t[j]$$

translates to

$$\begin{pmatrix} reg_o^1 = t.\texttt{reg} \wedge ts_o^1 = t.\texttt{ts} \wedge of_o^1 \leq of_o^2 \\ reg_o^2 = t.\texttt{reg} \wedge ts_o^2 = t.\texttt{ts} \wedge of_o^2 \leq n \end{pmatrix} \Rightarrow reg_d^{1,2} = 0 \wedge ts_d^{1,2} = 0 \wedge of_d^1 \leq of_d^2$$

4 Abstract semantics

Our abstract semantics can be computed using the standard set operators of Figure 5: join, meet, projection, assignments and guard. All off-the-shelf numeric domains provide abstraction for these operators. At first, the complement

$$
\begin{array}{lll}
\textit{Join} & d_1 \vee d_2 & \triangleq \{v \mid v \in d_1 \vee v \in d_2\} \\
\textit{Meet} & d_1 \wedge d_2 & \triangleq \{v \mid v \in d_1 \wedge v \in d_2\} \\
\textit{Projection} & \exists x.d & \triangleq \{v[x \mapsto n] \mid v \in d, n \in \mathbb{N}\} \\
\textit{Assignment} & d[x := e] & \triangleq \{v[x \mapsto [\![e]\!]_v] \mid v \in d\} \\
\textit{Guard} & e \diamond e' & \triangleq \{v \mid v[\![e]\!]_v \diamond [\![e']\!]_v\} \qquad \diamond \in \{=, >, \geq, \leq, <\} \\
\textit{Complement} & \neg d & \triangleq \{v \mid v \notin d\} \\
\textit{Implication} & d_1 \Rightarrow d_2 & \triangleq \neg d_1 \vee d_2
\end{array}
$$

Figure 5. Interface of numeric abstract domains

operator (and derived implication operator) might seem problematic. However, they are here more a notational convenience and all the abstract transformers can be rewritten without them by computing a *negative normal form*. This can always be done without loss of precision because i) set operations commute with respect to negation; ii) projections and assignments are never under a negation; iii) the language of guards is closed by negation. For a given numeric abstract domain, the best abstraction might not be obtained in a compositional way. Typically, numeric abstract domains feature a *test* operator which provides a more precise abstraction than computing the meet ($\sqcap$) of an abstract element with a guard ($\gamma(test(d, e \diamond e')) \subseteq \gamma(d \sqcap (e \diamond e'))$). To get more precise abstractions, the specification of abstract transformers given in the next section might require domain-specific massaging.

$$
\begin{aligned}
&[\![x^\iota := e]\!]^\sharp(d) && = d[x^\iota := e] && e \in e^\iota \setminus \{x^\pi.\texttt{len}, ?\} \\
&[\![x^\iota := ?]\!]^\sharp(d) && = \exists x^\iota.d \\
&[\![x^\pi := \texttt{null}]\!]^\sharp(d) && = d[x^\pi.\texttt{reg} := 0][x^\pi.\texttt{ts} := 0][x^\pi.\texttt{of} := 0] \\
&[\![x_1^\pi := x_2^\pi]\!]^\sharp(d) && = d[x_1^\pi.\texttt{reg} := x_2^\pi.\texttt{reg}][x_1^\pi.\texttt{ts} := x_1^\pi.\texttt{ts}][x_1^\pi.\texttt{of} := x_2^\pi.\texttt{of}] \\
&[\![x_1^\pi := x_2^\pi + x^\iota]\!]^\sharp(d) && = d[x_1^\pi.\texttt{reg} := x_2^\pi.\texttt{reg}][x_1^\pi.\texttt{ts} := x_2^\pi.\texttt{ts}][x_1^\pi.\texttt{of} := x_2^\pi.\texttt{of}+x^\iota] \\
&[\![x_1^\pi := x^\iota]\!]^\sharp(d) && = d[x_1^\pi.\texttt{reg} := 0][x_1^\pi.\texttt{ts} := 0][x_1^\pi.\texttt{of} := x^\iota]
\end{aligned}
$$

Figure 6. Standard abstract transformers

The formal definitions of the abstract transformers can be found in Figure 6 and Figure 7. Soundness proofs are rather systematic; they are mechanised in Coq [2] and the development is available for review.

Abstract transformers that do not access the heap are standard and can be found in Figure 6. The assignment of an integer variable is directly modelled by an abstract assignment and the assignment of a pointer variable x^π is modelled by an assignment of the variables $x^\pi.\texttt{reg}$, $x^\pi.\texttt{ts}$ and $x^\pi.\texttt{of}$. The definition of heap abstract transformers is more complicated and is presented in Figure 7. To express conditions about pointers and heap variables, we use the following notations:

$$
\begin{aligned}
orig^i(x^\pi.\alpha) &\triangleq x^\pi.\texttt{reg} = reg_o^i \wedge x^\pi.\texttt{ts} = ts_o^i \\
orig^i(x^\pi) &\triangleq x^\pi.\texttt{reg} = reg_o^i \wedge x^\pi.\texttt{ts} = ts_o^i \wedge x^\pi.\texttt{of} = of_o^i \\
dest^i(x^\pi) &\triangleq x^\pi.\texttt{reg} = reg_d^i \wedge x^\pi.\texttt{ts} = ts_d^i \wedge x^\pi.\texttt{of} = of_d^i \\
\exists_{x^\pi}.d &\triangleq \exists_{x^\pi.\texttt{reg}}\exists_{x^\pi.\texttt{ts}}\exists_{x^\pi.\texttt{of}}.d \qquad \exists_{\{x_1,\dots,x_n\}}.d \triangleq \exists_{x_1}.\dots.\exists_{x_n}.d
\end{aligned}
$$

For instance, the condition $orig^i(x^\pi.\alpha)$ states that the address of the pointer $x^\pi.\alpha$ is the address of the origin of the ith heap-slice. The abstract transfer function updating heap-slices are doing so using a *fold* iterator defined by

$$
\begin{aligned}
&fold : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp) \rightarrow \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp \\
&fold\ n\ f\ d = \begin{cases} d & \text{if n} = 0 \\ fold\ (n-1)\ f\ (f\ (n-1)\ d) \end{cases}
\end{aligned}
$$

Heap abstract transformers all follow a similar pattern and select the relevant heap-slices using the conditions $orig(x^\pi.\alpha)$ or $orig(x^\pi)$ thus allowing to query or (strong-)update heap-slices. In Figure 7, each abstract transformer is defined in two steps. First, we write a pre-transformer of the form $[\![stmt]\!]^{\sharp^i}$ which models the update or the query of the ith heap-slice. The final transformer $[\![stmt]\!]^\sharp$ is obtained by iterating the pre-transformer over the heap-slices. If the statement does not update the heap, the iterator is a conjunction of the form $\bigwedge_{i=1}^k [\![stmt]\!]^{\sharp^i}$; otherwise, it is a *fold* of the form $fold\ k\ (\lambda i.[\![stmt]\!]^{\sharp^i})$.

If the condition $orig(x^\pi.\alpha)$ holds then the value of the heap variable *size* is the length of the array pointed to by x^π. If the condition $orig(x^\pi)$ holds the content of x^π, *i.e.*, $*x^\pi$, is given by the values of the heap variables $((reg_d, ts_d), of_d)$.

$$\llbracket x^{\iota} := x^{\pi}.\mathtt{len} \rrbracket^{\sharp i}(d) \quad = \bigwedge \left(\begin{array}{l} orig^i(x^{\pi}.\alpha) \Rightarrow (size^i = x^{\iota}) \\ \exists_{HeapVar_i}.(\exists x^{\iota}.d \wedge orig^i(x^{\pi}.\alpha) \wedge size^i = x^{\iota}) \end{array} \right)$$

$$\llbracket x^{\iota} := x^{\pi}.\mathtt{len} \rrbracket^{\sharp}(d) \quad = \bigwedge_{i=1}^{k} \llbracket x^{\iota} := x^{\pi}.\mathtt{len} \rrbracket^{\sharp i}(\exists x^{\iota}.d)$$

$$\llbracket x_1^{\pi} := *x_2^{\pi} \rrbracket^{\sharp i}(d) \quad = \bigwedge \left(\begin{array}{l} orig^i(x_2^{\pi}) \Rightarrow dest^i(x_1^{\pi}) \\ \exists_{HeapVar_i}.(\exists x_1^{\pi}.d \wedge orig^i(x_2^{\pi}) \wedge dest^i(x_1^{\pi})) \end{array} \right)$$

$$\llbracket x_1^{\pi} := *x_2^{\pi} \rrbracket^{\sharp}(d) \quad = \bigwedge_{i=1}^{k} \llbracket x_1^{\pi} := *x_2^{\pi} \rrbracket^{\sharp i}(\exists x_1^{\pi}.d)$$

$$\llbracket *x_1^{\pi} := x_2^{\pi} \rrbracket^{\sharp i}(d) \quad = \bigwedge \left(\begin{array}{l} orig^i(x_1^{\pi}) \;\; \Rightarrow dest^i(x_2^{\pi}) \wedge \exists_{\{reg_d^i, ts_d^i, of_d^i\}}.d \\ \neg orig^i(x_1^{\pi}) \Rightarrow d \end{array} \right)$$

$$\llbracket *x_1^{\pi} := x_2^{\pi} \rrbracket^{\sharp}(d) \quad = fold \; k \; (\lambda i. \llbracket *x_1^{\pi} := x_2^{\pi} \rrbracket^{\sharp i}) \; d$$

$$\llbracket x^{\pi} := \mathtt{new}[x^{\iota}]^{\mathtt{r}} \rrbracket^{\sharp i}(d) = \bigwedge \left(\begin{array}{l} orig^i(x^{\pi}.\alpha) \;\; \Rightarrow size^i = x^{\iota} \wedge reg_d^i = 0 \wedge ts_d^i = 0 \wedge of_d^i = 0 \\ \qquad\qquad\qquad \wedge \exists_{\{size^i, reg_d^i, ts_d^i, of_d^i\}}.d \\ \neg orig^i(x^{\pi}.\alpha) \Rightarrow d \end{array} \right)$$

$$\llbracket x^{\pi} := \mathtt{new}[x^{\iota}]^{\mathtt{r}} \rrbracket^{\sharp}(d) \quad = \begin{array}{l} let \; d' := (\exists_{x^{\pi}}.d[r := r+1]) \wedge x^{\pi}.\mathtt{reg} = \mathtt{r} \wedge x^{\pi}.\mathtt{ts} = r \wedge x^{\pi}.\mathtt{of} = 0 \\ fold \; k \; \lambda i. \llbracket x^{\pi} := \mathtt{new}[x^{\iota}]^{\mathtt{r}} \rrbracket^{\sharp i} \; d' \end{array}$$

$$\llbracket \mathtt{free}(x^{\pi}) \rrbracket^{\sharp i}(d) \quad = \bigwedge \left(\begin{array}{l} orig^i(x^{\pi}.\alpha) \;\; \Rightarrow size^i = 0 \wedge reg_d^i = 0 \wedge ts_d^i = 0 \wedge of_d^i = 0 \\ \qquad\qquad\qquad \wedge \exists_{\{size^i, reg_d^i, ts_d^i, of_d^i\}}.d \\ \neg orig^i(x^{\pi}.\alpha) \Rightarrow d \end{array} \right)$$

$$\llbracket \mathtt{free}(x^{\pi}) \rrbracket^{\sharp}(d) \quad = fold \; k \; \lambda i. \llbracket \mathtt{free}(x^{\pi}) \rrbracket^{\sharp i} \; d$$

Figure 7. Heap abstract transformers

For the command $x_1^\pi := *x_2^\pi$, the expression $orig(x_2^\pi) \Rightarrow dest(x_1^\pi)$ says literally that if the pointer x_2^π has for address $((reg_o, ts_o), of_o)$ then the value of x_1^π is $((reg_d, ts_d), of_d)$. This information is not sufficient to prove the obvious fact that after the sequence

$$*x_2^\pi := y^\pi; x_1^\pi := *x_2^\pi$$

we have that $x_1^\pi = y^\pi$ – despite the fact that the abstract semantics of the command $*x_2^\pi := y^\pi$ also computes the conditional information $orig(x_2^\pi) \Rightarrow dest(y^\pi)$. The purpose of the expression $\exists_{HeapVar}.(\exists x_1^\pi.d \wedge orig(x_2^\pi) \wedge dest(x_1^\pi))$ is to exploit this conditional information. It makes the assumption that $orig(x_2^\pi) \wedge dest(x_1^\pi)$ and discharges this assumption by projecting away the heap variables. The logic for the abstract semantics of the command $x^\iota := x^\pi.\mathtt{len}$ is similar and relates the integer variable x^ι with the *size* heap variable.

The command $*x_1^\pi := x_2^\pi$ is a side-effect writing the value of x_2^π at the address pointed by x_1^π. For simplicity, the variables x_1^π and x_2^π are distinct. (This syntactic restriction can be circumvented by introducing an auxiliary variable.) The abstract semantics proceeds by case analysis and modifies only the heap-slice pointed by x_1^π. If $orig(x_1^\pi)$ holds then the heap-slice is updated; it is defined by $dest(x_2^\pi)$ and obsolete information about the destination is projected away. Otherwise $(\neg orig(x_1^\pi))$, the abstraction is kept unchanged.

The abstract semantics of the commands for allocating and freeing memory is almost the same. All the heap-slices such that the condition $orig(x^\pi)$ holds are initialised and all other heap-slices are unchanged. For the allocation command, the value of the region $\mathtt{r}$ is incremented and the x^π pointer is initialised such that its region $x^\pi.\mathtt{reg}$ is $\mathtt{r}$, its time-stamp $x^\pi.\mathtt{ts}$ is r (after incrementation) and its offset $x^\pi.\mathtt{of}$ is set to 0.

4.1 System of constraints

The abstract semantics $P^\sharp$ of a program P computes for each program label l an over-approximation of the possible stores and heaps reaching this label. The abstract semantics is defined as the least solution of a syntax-direct constraint system. For each concrete semantic rule of the form $(l, s, h) \rightarrow (l', s', h')$, we have a constraint of the form $[\![c]\!]^\sharp(P^\sharp(l)) \subseteq P^\sharp(l')$ where $[\![c]\!]^\sharp$ is the abstract transformer associated with the command at label l. The rules are given in Figure 8.

4.2 Discussion

We discuss below various extensions and variants of the concrete and abstract semantics that could be worth investigation.

First-order regions. In our semantics, each allocation occurs in a statically known region. This scheme could be extended by considering first-order regions such as those proposed by languages such as Cyclon [18] or Real-Time Java. We are confident that first-order regions do not pose any serious difficulty and could be modelled by just authorising region variables.

$$\frac{c \quad c \notin \left\{ \begin{array}{c} \texttt{assumez, ensurez,} \\ \texttt{goto, jz} \end{array} \right\}}{[\![c]\!]^{\sharp}(P^{\sharp}(l)) \subseteq P^{\sharp}(l+1)} \qquad \frac{\texttt{check}(x^{\iota}) \quad \texttt{check} \in \left\{ \begin{array}{c} \texttt{assumez,} \\ \texttt{ensurez} \end{array} \right\}}{P^{\sharp}(l) \wedge (x^{\iota} = 0) \subseteq P^{\sharp}(l+1)}$$

$$\frac{\texttt{goto } l'}{P^{\sharp}(l) \subseteq P^{\sharp}(l')} \qquad \frac{\texttt{jz}(x^{\iota}) \Rightarrow l'}{P^{\sharp}(l) \wedge (x^{\iota} = 0) \subseteq P^{\sharp}(l')} \qquad \frac{\texttt{jz}(x^{\iota}) \Rightarrow l'}{P^{\sharp}(l) \wedge (x^{\iota} \neq 0) \subseteq P^{\sharp}(l+1)}$$

Figure 8. Constraint generation

Inter-procedural analysis is usually a challenging issue for shape-analysis. For (relational) numeric analyses, an input-output relation can be computed by *cloning* the set of variables: the variable x_{pre} represents the value of the variable x at the beginning of the procedure while x_{curr} represents the current value of the same variable x. This approach should adapt to our context. This is however difficult to assess the precision of such abstraction.

Inference. As already mentioned, the abstract semantics can be computed using any numeric abstract domains using standard chaotic fixpoint iteration with widening and narrowing. However, precise analyses require disjunctive numeric abstract domains in order to track different alias hypotheses. Widening disjunctive domains is known to be challenging [1,28] and more work is needed to devise a widening capable of capturing typical alias patterns. In our examples, we put the emphasis on precision and were actually successful at proving with abstract invariants the functional correctness of the algorithms. More pragmatically, we intend to investigate coarser abstractions thus trading precision for scalability.

5 Formalisation and experiments

The correctness proofs of the abstract transformers presented in the previous section are mechanised [2] in the Coq proof assistant. We prove that any solution $P^{\sharp}$ of the abstract constraints given in Figure 8 is a sound over-approximation of the program semantics. The proofs are rather systematic and benefit in particular from decision procedures for arithmetic [3].

Though we do not prove the optimality of abstract transformers, we have a practical experimental assessment of the precision of the abstract semantics. Our Caml prototype takes as input a WHILE program annotated with (untrusted) abstract invariants and (executable) assertions. The assertions are compiled together with the program into the kernel language presented in Section 2. For each program, we prove that the abstract invariants are strong enough to show that the assertions are valid *i.e.*, for each command of the form $\texttt{ensurez}(x^{\iota})$ we prove that the variable x^{ι} has necessarily the value 0.

To prove the invariants, we generate abstract arithmetic verification conditions and discharge them using automatic provers. To fill in the gap between

specification and implementation, the abstract transformers of our Coq formalisation are given the syntactic form of formulae f with a context $\bullet$:

$$f ::= \bullet \mid e_1 \diamond e_2 \mid f_1 \vee f_1 \mid f_1 \wedge f_2 \mid \neg f \mid \exists x.f$$

Using this approach, the formalisation is slightly heavier. However, it has the advantage that the abstract transformers can be automatically compiled into Caml code by the extraction mechanism of Coq. Our abstract verification generator is using this possibility and therefore implements faithfully and provably the abstract semantics.

Our abstract invariants are all linear and all the program also any use assignments of linear expressions. Under these conditions, the abstract semantics ensures that verifications conditions are quantifier-free linear integer arithmetic formulae. Those formulae are automatically discharged by a SMT-solver [8] – we are using yices [12].

Certain abstract transformers double the size of the formulae. This explosion could be mitigated by simplifying formulae on-the-fly [10]. However, we are not aware of VcGen algorithms generating provably compact formulae [13]. For our experiments, despite the absence of any simplification our proof obligations are discharged in a few seconds. This might be a sign that the formulae are *easy* and contain redundancies.

We have annotated and automatically verified a collection of simple algorithms manipulating lists, arrays, matrices, combination of the above. We also have a first example constructing binary trees. For lists, we prove that we can construct, traverse and reverse singly-linked lists. Given the abstraction, doubly-linked lists algorithms should be equally easy to verify. For arrays, we are able to validate the 7 programs proposed by Halbwachs and Péron [16]. Among those, only array copy and insertion sort require to express the invariants using $\mathbb{D}_2^\sharp$. For matrices, we construct rectangular and triangular matrices. Our abstraction is also capable of coping with the motivating example of Gulwani and Tiwari [14]. This examples combines lists and arrays. This program also requires $\mathbb{D}_2^\sharp$.

Our Coq formalisation, Caml prototype and annotated programs are available at the address http://www.irisa.fr/celtique/fbesson/num-abs.html.

6 Related work

The dominant trend of shape-analysis is to combine a symbolic domain of graphs with a numeric domain [9,33,30,26,14,20,24,29]. Our abstract semantics is purely numeric and does not refer to a symbolic domain. Our memory model is based on a more concrete representation of addresses. This has the advantage that aliases are captured numerically. This has the side-effect that the same recursive data-structure, for instance a singly-linked list, might have different abstractions depending on the allocation pattern. First-order regions would give the programmer a handle on the allocation pattern and guide the abstraction.

Numeric abstractions of programs have been investigated by several works *e.g.*, [34,35,11,22,23]. A major difference with the abstract semantics presented

here is that they all rely on a preliminary alias or shape analysis. This simplifies the design but it is a well-known result of abstract-interpretation that more precise results are obtained by running analyses in parallel. Because our abstract semantics tracks aliases numerically, the cooperation between numeric analysis and the alias analysis is obtained for free.

Our abstract semantics is using a form of quantification that is encoded in the concretisation function. Halbwachs and Péron [16] propose an abstract domain that is using a limited form of quantification for analysing the content of integer arrays. Their domain is tuned for inference and unlike our abstract semantics can capture sorting algorithms. Their domain is incomparable with ours because our abstract semantics can state invariants about arbitrary arrays – in particular recursive arrays. Gulwani and Tiwari [14] propose a technique for inferring quantified invariants. Their logic is rich but the abstract operators are very heuristic. On the contrary, our abstract transformers are defined using the usual operators of numeric abstract domains. McCloskey *et al.* [24] extend the previous work by allowing different domains to collaborate by exchanging quantified facts. The collaboration scheme is sophisticated but requires annotations and does not ensure an optimal sharing of information between domains. Our abstract semantics is less expressive but the underlying theory is much simpler and the collaboration scheme between alias and numeric information is directly encoded into the abstract semantics.

Time-stamps semantics have been used by Might and Shivers [25] to improve the precision of control-flow analyses. The abstract domain of time-stamps is weak ($\{0, 1, +\infty\}$) but allows to model strong-updates when the time-stamp is 1 and (abstract) garbage-collection can reset the counter. Our analysis can use more powerful numeric (relational) abstract domains and models strong-updates in a general way. We have not investigated whether our numeric abstract semantics could accommodate for a form of abstract garbage-collection.

Venet [34] abstracts a time-stamp semantics for analysing array and recursive data-structures. A time-stamp is a n-uple which represents nested loop counters. This encoding has the advantage of carrying more information than a single time-stamp and being easier to abstract with standard numeric abstract domains. Only minor changes to our abstract semantics are needed to accomodate for this time-stamping strategy. The analysis is using a preliminary Steensgaard analysis [32] and Venet admits that "we can compute the shape of the graph and the numerical relations betwee timestamps simultaneously [. . .]. However, this would make the presentation of the analysis quite complicated." Our abstract semantics does such combinations for free and could therefore serve as a tool for reconstructing existing analyses in a common framework – each instance being obtained by specialising the numeric abstract domain.

7   Conclusion

In this paper, we propose an original approach for analysing heap-manipulating programs using a purely numeric abstract semantics. The abstraction uses a

time-stamp semantics and captures uniformly invariants of arrays and recursive data-structures. To our knowledge, the approach presented here has never been attempted before for general programs. Despite being extreme, the approach looks promising and is able to validate non-trivial invariants of programs using only (linear) decidable fragments of arithmetic. Our abstract invariants are inherently disjunctive and therefore their effective inference is challenging for current numeric abstract domains. More work is needed to assess the real difficulty of inferring our abstract invariants. From a more methodological standpoint, our abstraction could serve as a tool for reconstructing existing alias analyses and study the benefit of combining them with a numeric analysis. Because the analysis is already numeric, each analysis instance should be obtained by crafting a specific numeric abstract domain.

References

1. R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.
2. F. Besson. Mechanised soundness proof of a numeric abstract semantics. http://www.irisa.fr/celtique/fbesson/num-abs.
3. F. Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *TYPES*, volume 4502 of *LNCS*, pages 48–62. Springer, 2006.
4. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006.
5. L. Chen, A. Miné, J. Wang, and P. Cousot. Linear absolute value relation analysis. In *ESOP*, volume 6602 of *LNCS*, pages 156–175. Springer, 2011.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–97. ACM Press, 1978.
8. L. M. de Moura, B. Dutertre, and N. Shankar. A tutorial on satisfiability modulo theories. In *CAV*, volume 4590 of *LNCS*, pages 20–36. Springer, 2007.
9. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241. ACM press, 1994.
10. I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *SAS*, volume 6337 of *LNCS*, pages 236–252. Springer, 2010.
11. N. Dor, M. Rodeh, and S. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI*, pages 155–167. ACM, 2003.
12. B. Dutertre and L. De Moura. Yices: An SMT solver. Technical report, 2006.
13. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205. ACM, 2001.
14. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, volume 4590 of *LNCS*, pages 379–392. Springer, 2007.
15. A. Gurfinkel and S. Chaki. Boxes: A symbolic abstract domain of boxes. In *SAS*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010.
16. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

17. B. Jeannet, D. Gopan, and T. W. Reps. A relational abstraction for functions. In *SAS*, volume 3672 of *LNCS*, pages 186–202. Springer, 2005.
18. T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX*, pages 275–288, 2002.
19. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, pages 244–256. ACM, 1979.
20. V. Laviron, B-Y. E. Chang, and X. Rival. Separating shape graphs. In *ESOP*, volume 6012 of *LNCS*, pages 387–406. Springer, 2010.
21. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
22. S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.
23. S. Magill, M-H. Tsai, P. Lee, and Y-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, pages 211–222. ACM, 2010.
24. B. McCloskey, T. W. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, volume 6337 of *LNCS*, pages 71–99. Springer, 2010.
25. M. Might and O. Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ICFP*, pages 13–25. ACM, 2006.
26. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM Press, 2006.
27. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.
28. C. Popeea and W-N Chin. Inferring disjunctive postconditions. In *ASIAN*, volume 4435 of *LNCS*, pages 331–345. Springer, 2006.
29. X. Rival and B-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186. ACM, 2011.
30. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
31. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, volume 3385 of *LNCS*, pages 21–47. Springer, 2005.
32. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.
33. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program.*, 35(2):223–248, 1999.
34. A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *SAS*, volume 2477 of *LNCS*, pages 36–51. Springer, 2002.
35. A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS*, volume 3148 of *LNCS*. Springer, 2004.
36. P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *SAS*, volume 983 of *LNCS*, pages 21–32. Springer, 1995.