

Nonuniform Alias Analysis of Recursive Data Structures and Arrays^{*}

Arnaud Venet

LIX, École Polytechnique
91128 Palaiseau France
`venet@lix.polytechnique.fr`

Abstract. In this paper we present an alias analysis which is able to infer position-dependent equality relationships between pointers in recursively defined data structures or arrays. Our work is based on a semantic model of the execution of a program in which every allocated object is identified by a timestamp which represents the state of the program at the instant of the object creation. We provide a simple numerical abstraction of timestamps which is accurate enough to distinguish between elements of arrays or list-like structures. We follow the methodology of Abstract Interpretation to derive a sound approximation of the program semantics from this abstraction. The computability of our analysis is then guaranteed by using an abstract numerical lattice to represent relations between timestamps.

1 Introduction

Most program manipulation techniques, from compile-time optimization to automatic verification, rely on an alias analysis to ensure the soundness of their results. The precision of the analysis has a substantial impact on the effectiveness of the applications which use the aliasing information [15]. However, among the numerous alias analyses existing in the literature few can provide accurate information in the presence of recursive data structures [10–12, 21, 23, 24]. The analyses described in [12] and [21] can infer the shape of a structure (for example a list or a tree) and prove that this shape is preserved under certain operations. Deutsch [10, 11] has been the first one to propose a *nonuniform* alias analysis, that is an analysis where the aliasing relationships are parameterized by the position of the elements in recursive data structures. In a previous work [23, 24] we have shown how to extend the applicability and precision of Deutsch’s analysis by using a richer abstract domain. In this paper we present a new technique for computing position-dependent aliasing information which is both simpler and more expressive than the existing models.

The nonuniform alias analyses designed so far [10, 11, 23, 24] rely on a store-less model of the memory introduced by Jonkers [16] in which aliasing is explicitly

^{*} This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

<pre> void copy(List l1, List l2) { List p1, p2; p1 = l1; p2 = l2; while(p1 != null) { p1->val = p2->val; p1 = p1->next; p2 = p2->next; } } </pre>	<p>Let c be a counter for the loop, then:</p> $p1(->next)^i->val \equiv l1(->next)^j->val$ $p2(->next)^i->val \equiv l2(->next)^j->val$ <p style="text-align: center;">for all i, j such that $j = i + c$</p> <p>As a consequence of these relations:</p> $l1(->next)^i->val \equiv l2(->next)^j->val$ <p style="text-align: center;">for all i, j such that $j = i$</p> <p>After abstraction we have:</p> $p1(->next)^*->val \equiv l1(->next)^*->val$ $p2(->next)^*->val \equiv l2(->next)^*->val$ $l1(->next)^*->val \equiv l2(->next)^*->val$
--	--

Fig. 1: Limitation of analyses based on an aliasing relation

given by means of an equivalence relation over the access paths into the data structures manipulated by the program. The key idea of the abstraction is to represent the position of an element in a recursive structure by a tuple of counters denoting the number of times each recursive component of the structure has to be unfolded to give access to this element. Nonuniform aliasing relationships can then be expressed by using linear constraints on these counters. For instance, these analyses can discover that the n -th element of the list returned by a list copy function can only be aliased to n -th element of the list given as an argument to the function. However, this model has some drawbacks:

- Representing aliasing explicitly by a binary relation over access paths is costly (the number of alias pairs can increase quadratically by means of transitive closure) and redundant (if two elements in a structure are aliased, then all their common descendants are).
- The aliasing information remains precise as long as recursive structures are manipulated recursively, that is bottom-up. In the case of a list which is modified by an iterative top-down algorithm, these analyses will give poor results as illustrated in Fig. 1 where \equiv denotes the aliasing relation. The loss of precision comes from the fact that these analyses cannot represent relationships between more than two data structures. In the iterative list copy program of the figure, the aliasing relationships between $l1$ and $p1$ on one hand, and $l2$ and $p2$ on the other hand are related via an implicit loop counter. Since this information is lost by the abstraction, these analyses will compute a uniform approximation of the aliasing. The problem is that in practice list operations are implemented iteratively for efficiency purposes.
- There is no obvious way of representing array structures in these models since it would imply considering paths of data accessors which may contain integers, i.e. word languages over an infinite signature. It means that all classical algorithms over finite automata which underlie these analyses should be redesigned, which represents a considerable effort. This is a major restric-

tion, since arrays are pervasively used in common programming languages as object containers.

We propose to relieve these limitations by using a different semantic model as a basis for a nonuniform alias analysis which can cope with recursive data structures and arrays as well.

We start with a store-based semantic model in which the memory is described by a graph. Classically, in these models newly allocated objects are anonymously represented by fresh nodes which are added to the store. Instead, we explicitly identify each object with a timestamp which is an abstraction of the execution state at the moment of the object creation. This semantics of memory allocation was suggested in [9]. Burke, Carini, Choi and Hind [3] use a similar idea but they apply a k -limiting scheme which does not allow their analysis to distinguish between elements nested beyond depth k in a structure. In our model we abstract a timestamp by an integer which intuitively represents the number of times the corresponding allocation instruction has been executed in the current function call. This abstraction is precise enough to deal with common algorithms for iterating over a list or an array. We specify our analysis in the framework of Abstract Interpretation [5, 6, 4, 7] by deriving an approximation of the program semantics from an abstract representation of the memory.

In order to construct the abstract domain we need to know the shape of the memory graph. We use Steensgaard’s analysis [22] for simplicity, but this information can be provided by any extant pointer analysis. The edges of this graph are then decorated with numerical constraints relating the timestamps of the source and target nodes together with the element index in the case of arrays. These enriched storage graphs form our abstract representation of memory. Therefore, the abstract semantics simply amounts to performing arithmetic operations on integers. Array operations fit smoothly in this model and the analysis can infer nonuniform aliasing information relating arrays and recursive structures by means of numerical relations between timestamps. As far as we know, this is the first static analysis which can provide position-dependent aliasing information in the presence of arrays.

The paper is organized as follows. In Sect. 2 we define the syntax and semantics of the simple first-order programming language that we will use to describe our analysis. Section 3 presents the abstract interpretation framework which underlies the design of the analysis. In Sect. 4 we show how to construct the abstract domain upon the abstract storage graph provided by a prior static analysis. We sketch how to compute this abstract graph by using Steensgaard’s algorithm. Section 5 describes the abstract semantics of a program. We discuss extensions to this analysis in Sect. 6.

2 Definition of the Language

We will describe our analysis by using a simply-typed first-order imperative language. There are only two types of values in our language: integers, denoted by the type ι , and pointers, denoted by the type o . We assume that we are

provided with a set \mathcal{V} of variable names and a set \mathcal{F} of symbols denoting the fields of structured objects. A program P is a set of function declarations:

$$\mathbf{fun} f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = \mathbf{local} v_1 : \tau'_1, \dots, v_n : \tau'_k \mathbf{in} cmd$$

where the τ_i, τ'_j 's and τ are types, the x_i 's represent the function parameters and the v_i 's are local variables. A command cmd is defined by the following grammar:

$cmd ::= \{cmd ; \dots ; cmd\}$	sequential composition
$\mathbf{while} bool \mathbf{do} cmd$	iteration
$\mathbf{if} bool \mathbf{then} cmd \mathbf{else} cmd$	condition
$\mathbf{return} x$	function return
$x := f(x_1, \dots, x_n)$	function call
$x := \mathbf{malloc}$	memory allocation
$x[i] := y$	array update
$x.f := y$	field update
$x := y[i]$	array access
$x := y.f$	field access
$x := expr$	simple assignment

where $x, y, i, x_1, \dots, x_n \in \mathcal{V}$ and $f \in \mathcal{F}$. The syntax of expressions is defined as follows:

$expr ::= \mathbf{null}$	null reference
n	integer constant
$x + n$	incrementation
$bool ::= x < y \mid x < n \mid x = \mathbf{null}$	boolean expressions
$bool \ \& \ bool \mid \mathbf{not} \ bool$	

where $x, y \in \mathcal{V}$ and $n \in \mathbb{N}$. We assume that the variables appearing respectively on the lefthand and righthand sides of an assignment are distinct. The syntax is intentionally minimalist for the sake of clarity. We assume that programs are well-formed with respect to the usual context rules and well-typed with respect to types ι and o . We use a very loose representation of memory: an object does not hold any information about its type or size. All memory operations are valid except when a null pointer is involved. We represent the store by a graph in which the nodes are the allocated objects and the edges denote fields of structures or elements of arrays. Therefore, in this model an object can be both viewed as an array and a structure, and both kinds of operations apply. The purpose is to simplify the definition of the semantic rules.

The execution of a program P is modelled by a small-step operational semantics. We suppose that every command cmd in the program is uniquely labelled by $\ell : cmd$. We denote by \mathcal{L} the set of labels in P . For each $\ell : cmd$ where cmd is not a **return** command, we denote by $\text{next}(\ell)$ the label of the unique command which follows cmd in the control flow graph of the enclosing function. We denote by $\text{entry}(f)$ the label of the entry point of a function f , by $\text{params}(f)$ the tuple of formal parameters of f and by $\text{locals}(f)$ the set of local variables of f . We

$$\begin{array}{c}
\frac{}{\langle S, \ell : \{\ell' : \text{cmd} ; \dots\}, \varrho, M \rangle \rightarrow \langle S, \ell' : \text{cmd}, \varrho, M \rangle} \\
\frac{\llbracket b \rrbracket \varrho = \mathbf{true}, \varrho' = \varrho \{ \mathbf{c}_\ell \leftarrow \varrho(\mathbf{c}_\ell) + 1 \}}{\langle S, \ell : \mathbf{while } b \mathbf{ do } \ell' : \text{cmd}, \varrho, M \rangle \rightarrow \langle S, \ell' : \text{cmd}, \varrho', M \rangle} \\
\frac{\llbracket b \rrbracket \varrho = \mathbf{false}}{\langle S, \ell : \mathbf{while } b \mathbf{ do } \ell' : \text{cmd}, \varrho, M \rangle \rightarrow \langle S, \text{next}(\ell), \varrho, M \rangle} \\
\frac{\llbracket b \rrbracket \varrho = \mathbf{true}}{\langle S, \ell : \mathbf{if } b \mathbf{ then } \ell_1 : \text{cmd}_1 \mathbf{ else } \ell_2 : \text{cmd}_2, \varrho, M \rangle \rightarrow \langle S, \ell_1 : \text{cmd}_1, \varrho, M \rangle} \\
\frac{\llbracket b \rrbracket \varrho = \mathbf{false}}{\langle S, \ell : \mathbf{if } b \mathbf{ then } \ell_1 : \text{cmd}_1 \mathbf{ else } \ell_2 : \text{cmd}_2, \varrho, M \rangle \rightarrow \langle S, \ell_2 : \text{cmd}_2, \varrho, M \rangle} \\
\frac{S = S'.(\ell' : y := f(x_1, \dots, x_n), \varrho')}{\langle S, \ell : \mathbf{return } x, \varrho, M \rangle \rightarrow \langle S', \text{next}(\ell'), \varrho' \{ y \leftarrow \varrho(x) \}, M \rangle} \\
\text{for all } x \in \text{locals}(f) : \varrho'(x) = \begin{cases} \perp_{\mathcal{T}} & \text{if } x : o \\ 0 & \text{if } x : \iota \end{cases}, \text{params}(f) = (p_1, \dots, p_n), \\
\text{for all } 1 \leq i \leq n : \varrho'(p_i) = \varrho(x_i), \text{for all } \ell \in \text{loops}(f) : \varrho'(\mathbf{c}_\ell) = 0 \\
\hline
\langle S, \ell : y := f(x_1, \dots, x_n), \varrho, M \rangle \rightarrow \langle S.(\ell, \varrho), \text{entry}(f), \varrho', M \rangle
\end{array}$$

Fig. 2: Concrete semantics of the language: control structures.

choose a model of the memory suggested by Deutsch [9] where all objects are uniquely identified with a timestamp which is an image of the execution state at the instant of the object creation. The sequence of return points occurring in the call stack at the execution of a **malloc** seems the most obvious choice for timestamps. However, this information is not sufficient to identify uniquely an object because of loops. The idea is to enrich this timestamp by attaching to each return point in the call stack the number of times every loop of the enclosing function has been executed. Then we can unambiguously refer to an object via its timestamp.

More formally, we denote by $\text{loops}(f)$ the set of labels of all **while** loops occurring in the function f . We associate to each $\ell \in \text{loops}(f)$ a distinct counter \mathbf{c}_ℓ in \mathcal{V} . A timestamp is a sequence $(\ell_1, \varrho_1) \dots (\ell_n, \varrho_n)$ where, for each $1 \leq i \leq n$, ℓ_i is the label of a function call, and ϱ_i is an environment mapping every loop counter of the function containing ℓ_i to an integer. We denote by $\perp_{\mathcal{T}}$ the special timestamp representing a null pointer. Let \mathcal{T} be the set of all timestamps. We denote by $\text{vars}(f)$ the set of local variables, formal parameters and loop counters of the function f . A state s of the operational semantics is a tuple $\langle S, \ell, \varrho, M \rangle$,

$$\begin{array}{c}
\frac{\sigma = \text{stamp}(S.(\ell, \varrho))}{\langle S, \ell : x := \mathbf{malloc}, \varrho, (N, E) \rangle \rightarrow \langle S, \text{next}(\ell), \varrho\{x \leftarrow \sigma\}, (N \cup \{\sigma\}, E) \rangle} \\
\frac{\varrho(x) = \perp_{\mathcal{T}}}{\langle S, \ell : \begin{cases} x.f := y \\ x[i] := y \end{cases}, \varrho, M \rangle \rightarrow \Omega} \\
\frac{\varrho(x) \neq \perp_{\mathcal{T}}, E' = E - (\{\varrho(x)\} \times \{f\} \times S)}{\langle S, \ell : x.f := y, \varrho, (N, E) \rangle \rightarrow \langle S, \text{next}(\ell), \varrho, (N, E' \cup \{(\varrho(x), f, \varrho(y))\}) \rangle} \\
\frac{\varrho(x) \neq \perp_{\mathcal{T}}, E' = E - (\{\varrho(x)\} \times \{\varrho(i)\} \times \mathcal{T})}{\langle S, \ell : x[i] := y, \varrho, (N, E) \rangle \rightarrow \langle S, \text{next}(\ell), \varrho, (N, E' \cup \{(\varrho(x), \varrho(i), \varrho(y))\}) \rangle} \\
\frac{\varrho(y) = \perp_{\mathcal{T}}}{\langle S, \ell : \begin{cases} x := y.f \\ x := y[i] \end{cases}, \varrho, M \rangle \rightarrow \Omega} \\
\frac{\varrho(y) \neq \perp_{\mathcal{T}}, \sigma = \begin{cases} \sigma' & \text{if } (\varrho(y), f, \sigma') \in E \\ \perp_{\mathcal{T}} & \text{otherwise} \end{cases}}{\langle S, \ell : x := y.f, \varrho, (N, E) \rangle \rightarrow \langle S, \text{next}(\ell), \varrho\{x \leftarrow \sigma\}, (N, E) \rangle} \\
\frac{\varrho(y) \neq \perp_{\mathcal{T}}, \sigma = \begin{cases} \sigma' & \text{if } (\varrho(y), \varrho(i), \sigma') \in E \\ \perp_{\mathcal{T}} & \text{otherwise} \end{cases}}{\langle S, \ell : x := y[i], \varrho, (N, E) \rangle \rightarrow \langle S, \text{next}(\ell), \varrho\{x \leftarrow \sigma\}, (N, E) \rangle} \\
\frac{}{\langle S, \ell : x := e, \varrho, M \rangle \rightarrow \langle S, \text{next}(\ell), \varrho\{x \leftarrow \llbracket e \rrbracket \varrho\}, M \rangle}
\end{array}$$

Fig. 3: Concrete semantics of the language: assignment commands.

where ℓ is the label of a command, ϱ is an environment mapping each $x \in \text{vars}(f)$, where f is the function enclosing the command labelled by ℓ , to a timestamp in \mathcal{T} if x has type o , to an integer otherwise. The component S of the state s is a call stack, that is a sequence $(\ell_1, \varrho_1) \dots (\ell_n, \varrho_n)$ where, for all $1 \leq i \leq n$, ℓ_i is the label of a function call and ϱ_i is an environment at that point. The component M of s is a memory configuration, that is a labelled directed graph (V, E) , where $V \subseteq \mathcal{T}$ and $E \subseteq V \times (\mathcal{F} \cup \mathbb{N}) \times V$. We denote by \mathcal{M} the set of all memory configurations. We also consider a special error state Ω which is used to model runtime errors. We denote by Σ the set of all states s . The operational semantics is then given by a transition relation $\rightarrow \in \varrho(\Sigma \times \Sigma)$.

If ϱ is an environment, $A \subseteq \mathcal{V}$, $x \in \mathcal{V}$ and $v \in \mathcal{T} \uplus \mathbb{N}$, we denote by $\varrho|_A$ the restriction of the domain of ϱ to the variables of A , and by $\varrho\{x \leftarrow v\}$ the environ-

ment which maps x to v and every variable $y \neq x$ to $\varrho(y)$. If $S = (\ell_1, \varrho_1) \dots (\ell_n, \varrho_n)$ is a call stack and, for all $1 \leq i \leq n$, f_i is the function enclosing the command labelled by ℓ_i , we define the timestamp corresponding to S as follows:

$$\text{stamp}(S) = (\ell_1, \varrho_1 |_{\text{loops}(f_1)}) \cdots (\ell_n, \varrho_n |_{\text{loops}(f_n)})$$

The transition relation of the small-step operational semantics is then defined by a set of rules in Fig. 2 and Fig. 3. Notice that whenever a function is called, all local variables are initialized to 0 or $\perp_{\mathcal{T}}$ accordingly to their type. The semantics $\llbracket e \rrbracket$ of an expression maps an environment ϱ to a value of the corresponding type as follows, boolean expressions being interpreted over the usual boolean algebra (**true**, **false**, \wedge , \vee , \neg):

$$\begin{array}{ll} \llbracket \mathbf{null} \rrbracket \varrho = \perp_{\mathcal{T}} & \llbracket x < y \rrbracket \varrho = (\varrho(x) < \varrho(y)) \\ \llbracket n \rrbracket \varrho = n & \llbracket x < n \rrbracket \varrho = (\varrho(x) < n) \\ \llbracket x + n \rrbracket \varrho = \varrho(x) + n & \llbracket e_1 \ \& \ e_2 \rrbracket \varrho = \llbracket e_1 \rrbracket \varrho \wedge \llbracket e_2 \rrbracket \varrho \\ \llbracket x = \mathbf{null} \rrbracket \varrho = (\varrho(x) = \perp_{\mathcal{T}}) & \llbracket \mathbf{not} \ e \rrbracket \varrho = \neg \llbracket e \rrbracket \varrho \end{array}$$

We assume that there is an entry function *main* in the program P which takes no arguments. The initial state i_P of the program is then defined as follows:

$$i_P = \langle \varepsilon, \text{entry}(\text{main}), \varrho_i, (\emptyset, \emptyset) \rangle$$

where ϱ_i maps every local variable of *main* to 0 or $\perp_{\mathcal{T}}$ accordingly to its type. Our purpose is to analyze the *collecting semantics* \mathcal{S} of P [4] which consists of all the states reachable from i_P by the transition relation:

$$\mathcal{S} = \{s \in \Sigma \mid i_P \xrightarrow{*} s\}$$

Example 1. We consider the following program:

```

make(n :  $\iota$ ) : o =
  local i :  $\iota$ , cell : o,
    p : o, val : o in {
11: while n > 0 do {
12:   cell := malloc ;
13:   val := malloc ;
14:   cell.next := p ;
15:   cell.value := val ;
16:   p := cell ;
17:   i := n - 1 ;
18:   n := i
   } ;
19: return p
}

copy(l : o) : o =
  local i :  $\iota$ , j :  $\iota$ ,
    t : o, m : o {
110: t := malloc ;
111: while not(l = null) do {
112:   t[i] := l.val ;
113:   m := l.next ;
114:   l := m ;
115:   j := i + 1 ;
116:   i := j ;
   } ;
117: return t
}

```

```

main() :  $\iota$  =
  local l : o, t : o in {
118:   l := make (10) ;
119:   t := copy(l) ;
120:   return 0
  }

```

The function `make` creates a list in a bottom-up way and the function `copy` stores pointers to the elements in an array by a top-down traversal of the list. At program point 120 we have:

$$\begin{aligned} \varrho(l) &= (118, \{\}) \cdot (12, \{\mathbf{c}_{11} \mapsto 1\}) \\ \varrho(t) &= (119, \{\}) \cdot (110, \{\mathbf{c}_{111} \mapsto 0\}) \end{aligned}$$

The aliasing between the list and the array are expressed by the following edges in the memory graph:

$$\begin{aligned} &((118, \{\}) \cdot (12, \{\mathbf{c}_{11} \mapsto i\}), \text{value}, (118, \{\}) \cdot (13, \{\mathbf{c}_{11} \mapsto i\})) \\ &((118, \{\}) \cdot (12, \{\mathbf{c}_{11} \mapsto j+1\}), \text{next}, (118, \{\}) \cdot (12, \{\mathbf{c}_{11} \mapsto j\})) \\ &((118, \{\}) \cdot (12, \{\mathbf{c}_{11} \mapsto 1\}), \text{next}, \perp_{\mathcal{T}}) \\ &((119, \{\}) \cdot (110, \{\mathbf{c}_{111} \mapsto 0\}), i, (118, \{\}) \cdot (13, \{\mathbf{c}_{11} \mapsto i\})) \end{aligned}$$

for $1 \leq i \leq 10, 1 \leq j \leq 9$. □

We use the methodology of Abstract Interpretation to build a sound approximation of the collecting semantics.

3 Abstract Interpretation

The collecting semantics can be expressed as the least fixpoint of the the complete \cup -morphism \mathbb{F} defined over the lattice $(\wp(\Sigma), \emptyset, \cup, \Sigma, \cap)$ as follows [4]:

$$\forall X \in \wp(\Sigma) : \mathbb{F}(X) = \{i_P\} \cup \{s \in \Sigma \mid \exists s' \in X : s' \rightarrow s\}$$

In general the iterative computation of this fixpoint does not terminate and \mathcal{S} is not finitely representable.

Abstract Interpretation [5, 6, 4, 7] offers various constructive frameworks for computing a safe approximation of \mathcal{S} . In this paper we follow the methodology defined in [7] which consists of designing an *abstract semantic specification* $(D^\sharp, \sqsubseteq, \perp^\sharp, \gamma, \mathbb{F}^\sharp, \sqcup, \nabla)$ where (D^\sharp, \sqsubseteq) is a partially ordered set, the *abstract domain*, related to concrete states by a monotonic operator $\gamma : (D^\sharp, \sqsubseteq) \rightarrow (\wp(\Sigma), \subseteq)$, the *concretization function*. The abstract semantics is described by a function $\mathbb{F}^\sharp : D^\sharp \rightarrow D^\sharp$, satisfying the following soundness condition:

$$\mathbb{F} \circ \gamma \subseteq \gamma \circ \mathbb{F}^\sharp$$

The element \perp^\sharp of D^\sharp provides us with an abstraction of the initial state, that is $\{i_P\} \subseteq \gamma(\perp^\sharp)$. The *join* $\sqcup : D^\sharp \times D^\sharp \rightarrow D^\sharp$ is an associative and commutative operation which computes an upper-bound of two elements in D^\sharp , and the *widening* $\nabla : D^\sharp \times D^\sharp \rightarrow D^\sharp$ is a binary operation over D^\sharp which satisfies the following conditions:

1. $\forall x, y \in D^\sharp : x \sqsubseteq x \nabla y, y \sqsubseteq x \nabla y$
2. For every increasing sequence $(x_n)_{n \geq 0}$ of elements of D^\sharp , the sequence $(y_n)_{n \geq 0}$ defined as follows:

$$\begin{cases} y_0 &= x_0 \\ y_{n+1} &= y_n \nabla x_{n+1} \end{cases}$$

is ultimately stationary.

Intuitively ∇ can be seen as an upper-bound operation which entails convergence when applied repeatedly.

We use this abstract semantic specification to compute the *abstract iteration sequence* $(\mathbb{F}_n^\sharp)_{n \geq 0}$ which mimicks the iterative fixpoint computation of \mathbb{F} :

$$\begin{cases} \mathbb{F}_0^\sharp &= \perp^\sharp \\ \mathbb{F}_{n+1}^\sharp &= \mathbb{F}_n^\sharp & \text{if } \mathbb{F}^\sharp(\mathbb{F}_n^\sharp) \sqsubseteq \mathbb{F}_n^\sharp \\ & \mathbb{F}_n^\sharp \nabla \mathbb{F}^\sharp(\mathbb{F}_n^\sharp) & \text{otherwise} \end{cases}$$

Theorem 1 (Cousot & Cousot[7]). *The sequence $(\mathbb{F}_n^\sharp)_{n \geq 0}$ is ultimately stationary. If $N \in \mathbb{N}$ is such that $\mathbb{F}_{N+1}^\sharp = \mathbb{F}_N^\sharp$, then for all $n \geq N$, $\mathbb{F}_n^\sharp = \mathbb{F}_N^\sharp$ and $\mathcal{S} \subseteq \gamma(\mathbb{F}_N^\sharp)$.*

This theorem provides us with an effective algorithm for computing an approximation of the collecting semantics \mathcal{S} . We will apply this scheme to our alias analysis. First, we construct the abstract domain specification, that is the tuple $(D^\sharp, \sqsubseteq, \perp^\sharp, \gamma, \sqcup, \nabla)$.

4 Construction of the Abstract Domain Specification

The whole abstract specification will rely on the approximation of timestamps. We choose a simple abstraction which only retains the total number of iterations in the topmost environment of a timestamp. More formally, let $\sigma = (\ell_{1,e} \text{ nv}_1) \dots (\ell_n, \varrho_n) \in \mathcal{T}$ be a timestamp. Let f be the function enclosing the command labelled by ℓ_n . The abstraction $\alpha_{\mathcal{T}}(\sigma)$ of σ is then defined as follows:

$$\alpha_{\mathcal{T}}(\sigma) = \sum_{\ell \in \text{loops}(f)} \varrho_n(\mathbf{c}_\ell)$$

The null pointer reference is abstracted by $\alpha_{\mathcal{T}}(\perp_{\mathcal{T}}) = 0$. This abstraction loses all information about the shape of the stack and the iteration counters in the callers. However, it is precise enough to capture common iterative object manipulations which are performed within a same function. Moreover the associated semantic operations are fairly simple as we will see in the next section. The specification of our analysis can be readily adapted to more complex timestamp abstractions.

The abstraction of the environments and the memory will be based on relations between integers. Therefore we need a computable representation of tuples of integers. Numerous abstract numerical domains have been developed for this purpose [17, 8, 13, 14, 18, 19] with various levels of expressiveness. We

leave the choice of such a domain as a parameter of our abstract specification. Following [11], we give a characterization of the primitive operations that an abstract numerical domain implements and we construct the semantic specifications upon these operations. Therefore, choosing a particular numerical domain merely amounts to instantiating these basic operations. More precisely, an abstract numerical domain is a collection of abstract domain specifications $(N_V^\sharp, \sqsubseteq_V, \perp_V^\sharp, \gamma_V : N_V^\sharp \rightarrow \wp(\mathbb{N}^V), \sqcup_V, \nabla_V)$ indexed by finite sets of variables $V \subseteq \mathcal{V}$. We denote by \top_V^\sharp an element of N_V^\sharp such that $\gamma_V(\top_V^\sharp) = \mathbb{N}^V$. The primitive operations are characterized as follows:

- If S is a system of linear equality constraints over V , the operation **add** $S : N_V^\sharp \rightarrow N_V^\sharp$ satisfies the following condition:

$$\forall \varrho^\sharp \in N_V^\sharp : \gamma_V(\mathbf{add} S(\varrho^\sharp)) \supseteq \{\varrho \in \gamma_V(\varrho^\sharp) \mid \varrho \text{ is a solution of } S\}$$

- If $x_1, \dots, x_n \in V$, the operation **delete** $_{x_1, \dots, x_n} : N_V^\sharp \rightarrow N^\sharp$ satisfies the following condition:

$$\forall \varrho^\sharp \in N_V^\sharp : \gamma_V(\mathbf{delete}_{x_1, \dots, x_n}(\varrho^\sharp)) \supseteq \{\varrho \in \mathbb{N}^V : \exists \varrho' \in \gamma_V(\varrho^\sharp) : \varrho'|_{V - \{x_1, \dots, x_n\}} = \varrho|_{V - \{x_1, \dots, x_n\}}\}$$

- If $x_1, \dots, x_n \in V$, the operation **project** $_{x_1, \dots, x_n} : N_V^\sharp \rightarrow N_{\{x_1, \dots, x_n\}}^\sharp$ satisfies the following condition:

$$\forall \varrho^\sharp \in N_V^\sharp : \gamma_{\{x_1, \dots, x_n\}}(\mathbf{project}_{x_1, \dots, x_n}(\varrho^\sharp)) \supseteq \{\varrho|_{\{x_1, \dots, x_n\}} \mid \varrho \in \gamma_V(\varrho^\sharp)\}$$

- If $\varrho_1^\sharp \in N_V^\sharp$ and $\varrho_2^\sharp \in N_W^\sharp$, then **mix** $(\varrho_1^\sharp, \varrho_2^\sharp)$ is an element of $N_{V \cup W}^\sharp$ which satisfies the following condition:

$$\begin{aligned} \forall \varrho_1 \in \gamma_V(\varrho_1^\sharp) : \forall \varrho_2 \in \gamma_W(\varrho_2^\sharp) : \forall \varrho \in \mathbb{N}^{V \cup W} : \\ (\varrho|_V = \varrho_1 \wedge \varrho|_W = \varrho_2) \Rightarrow \varrho \in \gamma_{V \cup W}(\mathbf{mix}(\varrho_1^\sharp, \varrho_2^\sharp)) \end{aligned}$$

Example 2. We consider the abstract numerical domain of Karr [17]. In this domain an element ρ^\sharp of N_V^\sharp is given by a system of linear equality constraints over the variables of V . This system is kept in row-echelon normal form by using Gauss algorithm. The operation **add** $S(\rho^\sharp)$ simply corresponds to adding the equations of S to the system ρ^\sharp and to normalize the resulting system. The operations **delete** $_{x_1, \dots, x_n}$ consists of eliminating all the constraints involving the variables x_1, \dots, x_n . The operation **project** is dual to **delete**, i.e. we keep all the constraints involving the given variables. The operation **mix** $(\rho_1^\sharp, \rho_2^\sharp)$ consists of taking the union of the systems ρ_1^\sharp and ρ_2^\sharp . Karr's domain satisfies the ascending chain condition, therefore we can use the join operation \sqcup_V as a widening. \square

For each label $\ell \in \mathcal{L}$, if f is the function enclosing the command labelled by ℓ , we denote by $\text{vars}@l$ the set of variables $\text{vars}(f)$. We suppose that we have an abstract domain specification for memory configurations $(\mathcal{M}^\sharp, \sqsubseteq_{\mathcal{M}}, \perp_{\mathcal{M}}^\sharp, \gamma_{\mathcal{M}} : \mathcal{M}^\sharp \rightarrow \wp(\mathcal{M}), \sqcup_{\mathcal{M}}, \nabla_{\mathcal{M}})$. Then, the abstract domain D^\sharp is defined as follows:

$$D^\sharp = \mathcal{L} \rightarrow \left(N_{\text{vars}@l}^\sharp \times M^\sharp \right)$$

The order relation \sqsubseteq over D^\sharp is the pointwise extension of the order relations over the abstract numerical domains and M^\sharp . The base element \perp^\sharp is given by $\langle \varrho_{\text{vars}@l}^\perp, \perp_{\mathcal{M}}^\sharp \rangle_{l \in \mathcal{L}}$ where:

$$\varrho_{\text{vars}@l}^\perp = \begin{cases} \mathbf{add} \{v = 0 \mid v \in \text{vars}@l\}(\top_{\text{vars}@l}^\sharp) & \text{when } l = \text{entry}(\text{main}) \\ \perp_{\text{vars}@l}^\sharp & \text{otherwise.} \end{cases}$$

If $s^\sharp \in D^\sharp$, we denote by $\langle \varrho(s^\sharp@l), M(s^\sharp@l) \rangle$ the couple $s^\sharp(l)$. The concretization function is then defined as follows:

$$\forall s^\sharp \in D^\sharp : \\ \gamma(s^\sharp) = \{\Omega\} \cup \{(S, \ell, \varrho, M) \mid \varrho \in \gamma_{\text{vars}@l}(\varrho(s^\sharp@l)) \wedge M \in \gamma_{\mathcal{M}}(M(s^\sharp@l))\}$$

This corresponds to the usual partitioning of execution states with respect to the control points [4]. Note that for the sake of simplicity we do not try to handle runtime errors precisely. The widening ∇ and the join \sqcup are defined by pointwise application of the corresponding operators over the component domains.

Now it remains to define the abstract domain specification for memory configurations. Intuitively, an abstract memory configuration is a storage graph in which edges carry relations between the timestamps of the source and target nodes. However, in order to construct this domain we need to know the shape of the memory graph. We can compute the shape of the graph and the numerical relations between timestamps simultaneously, by using the technique of *cofibred domains* [23, 24] and an associated widening operator. However, this would make the presentation of the analysis quite complicated. For explanatory purposes, we choose a simpler solution which consists of applying a flow-insensitive pointer analysis prior to our analysis which can infer an abstract storage graph describing the shape of the memory and the values of each pointer variable at every control point. We use the graph produced by this preliminary analysis to construct the abstract domain. We suppose that the results of this analysis are provided in the following form:

- A graph $G^\sharp = (V^\sharp, E^\sharp)$ where V^\sharp is a set of nodes and $E^\sharp \subseteq V^\sharp \times (\mathcal{F} \uplus \{\square\}) \times V^\sharp$, \square being a special symbol denoting array elements.
- A function μ mapping every couple (ℓ, x) where $x \in \text{vars}@l$ has type o , to a vertex in G^\sharp .

Moreover, G^\sharp must be deterministic: $\forall (s, l, t), (s', l', t') \in E^\sharp : (s = s' \wedge l = l') \Rightarrow t = t'$. The abstract graph G^\sharp must also satisfy the following soundness conditions:

$$\forall (S, \ell, \varrho, (V, E)) \in \mathcal{S} : \exists \nu : \mathcal{T} \rightarrow V^\sharp : \begin{cases} \forall x \in \text{vars}@l : (x : o) \Rightarrow \nu(\varrho(x)) = \mu(\ell, x) \\ \forall (\sigma_1, f, \sigma_2) \in E : (\nu(\sigma_1), f, \nu(\sigma_2)) \in E^\sharp \\ \forall (\sigma_1, n, \sigma_2) \in E : (\nu(\sigma_1), \square, \nu(\sigma_2)) \in E^\sharp \end{cases}$$

We distinguish three variables $\mathbf{s}, \mathbf{t}, \mathbf{i}$ which will respectively denote the abstract timestamp of the source vertex, the abstract timestamp of the target vertex and

the index of an array element for every edge of G^\sharp . Then, we define the abstract domain \mathcal{M}^\sharp of memory configurations as:

$$\mathcal{M}^\sharp = \left(\prod_{(s,f,t) \in E^\sharp} N_{\{s,t\}}^\sharp \right) \times \left(\prod_{(s,\square,t) \in E^\sharp} N_{\{s,i,t\}}^\sharp \right)$$

The order relation $\sqsubseteq_{\mathcal{M}}$ is the pointwise extension of the corresponding order relations over the abstract numerical domains. The widening $\nabla_{\mathcal{M}}$ and the join $\sqcup_{\mathcal{M}}$ are defined similarly. The abstract memory configuration $\perp_{\mathcal{M}}$ maps every vertex in E^\sharp to the base element \perp^\sharp of the corresponding numerical lattice. The concretization $\gamma_{\mathcal{M}}(M^\sharp)$ of an abstract memory configuration is the set of memory graphs (V, E) such that there exists a mapping $\nu : \mathcal{T} \rightarrow V^\sharp$ satisfying the following conditions:

1. $\forall (\sigma_1, f, \sigma_2) \in E : \langle s \mapsto \alpha_{\mathcal{T}}(\sigma_1), \mathbf{t} \mapsto \alpha_{\mathcal{T}}(\sigma_2) \rangle \in \gamma_{\{s,t\}}(M^\sharp(\nu(\sigma_1), f, \nu(\sigma_2)))$
2. $\forall (\sigma_1, n, \sigma_2) \in E :$
 $\langle s \mapsto \alpha_{\mathcal{T}}(\sigma_1), \mathbf{i} \mapsto n, \mathbf{t} \mapsto \alpha_{\mathcal{T}}(\sigma_2) \rangle \in \gamma_{\{s,i,t\}}(M^\sharp(\nu(\sigma_1), \square, \nu(\sigma_2)))$

This completes the specification of the abstract domain. Note that, even though the preliminary pointer analysis is flow-insensitive, our analysis is flow-sensitive. The shape of the abstract storage graph is the same at all control points, but the annotations which relate the timestamps associated to the nodes of this graph do depend on the control flow.

Steensgaard [22] designed a points-to analysis which can compute the graph G^\sharp in almost-linear time. The algorithm consists of assigning a distinct vertex to each variable of the program. The analysis then constructs an equivalence relation over these vertices, the edges of the graph linking the cosets of this equivalence relation. Every statement in the program is analyzed once resulting into the identification of vertices and/or the addition of new edges. The use of a union-find structure to represent the equivalence relation over the vertices makes this algorithm extremely efficient. We required that the abstract storage graph G^\sharp be deterministic since it simplifies the presentation of our analysis. This implies the use of unification-based pointer analyses like Steensgaard's but it is in no way an intrinsic limitation of our model. We could have formulated the construction of our abstract domain by using an inclusion-based pointer analysis like Andersen's [1] as well.

Example 3. The application of Steensgaard's pointer analysis to the program defined in Example 1 produces the following abstract graph:

$$G^\sharp = (\{n_1, n_2, n_3\}, \{(n_1, \mathbf{next}, n_1), (n_1, \mathbf{value}, n_2), (n_3, \square, n_2)\})$$

together with the corresponding assignment of nodes to variables:

$$\{\mathbf{t} \mapsto n_3, \mathbf{l} \mapsto n_1, \mathbf{m} \mapsto n_1, \mathbf{cell} \mapsto n_1, \mathbf{p} \mapsto n_1, \mathbf{val} \mapsto n_2\}$$

□

-
1. $\llbracket x := \mathbf{malloc} \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle [\mathbf{delete}_x ; \mathbf{add} \{x = \sum_{\ell \in \text{lo ops}(f)} \mathbf{c}_\ell\}] (\varrho^\sharp), M^\sharp \rangle$
 2. $\llbracket x[i] := y \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle \varrho^\sharp, M^\sharp \{e^\sharp \leftarrow M^\sharp(e^\sharp) \sqcup_{\{s,i,t\}} T(\varrho^\sharp)\} \rangle$ where
 - $e^\sharp = (\mu(x), \square, \mu(y))$
 - $T = [\mathbf{add} \{s = x, i = i, t = y\} ; \mathbf{project}_{s,i,t}]$
 3. $\llbracket x.f := y \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle \varrho^\sharp, M^\sharp \{e^\sharp \leftarrow M^\sharp(e^\sharp) \sqcup_{\{s,t\}} T(\varrho^\sharp)\} \rangle$ where
 - $e^\sharp = (\mu(x), f, \mu(y))$
 - $T = [\mathbf{add} \{s = x, t = y\} ; \mathbf{project}_{s,t}]$
 4. $\llbracket \ell : x := y[i] \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle T(M^\sharp(e^\sharp), \varrho^\sharp), M^\sharp \rangle$ where
 - $e^\sharp = (\mu(x), \square, \mu(y))$
 - $T = [\mathbf{mix} ; \mathbf{delete}_x ; \mathbf{add} \{s = y, i = i, t = x\} ; \mathbf{project}_{\text{vars}@\ell}]$
 5. $\llbracket \ell : x := y.f \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle T(M^\sharp(e^\sharp), \varrho^\sharp), M^\sharp \rangle$ where
 - $e^\sharp = (\mu(x), f, \mu(y))$
 - $T = [\mathbf{mix} ; \mathbf{delete}_x ; \mathbf{add} \{s = y, t = x\} ; \mathbf{project}_{\text{vars}@\ell}]$
 6. $\llbracket x := \mathbf{null} \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle [\mathbf{delete}_x ; \mathbf{add} \{x = 0\}] (\varrho^\sharp), M^\sharp \rangle$
 7. $\llbracket x := n \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle [\mathbf{delete}_x ; \mathbf{add} \{x = n\}] (\varrho^\sharp), M^\sharp \rangle$
 8. $\llbracket x := y + n \rrbracket^\sharp \langle \varrho^\sharp, M^\sharp \rangle = \langle [\mathbf{delete}_x ; \mathbf{add} \{x = y + n\}] (\varrho^\sharp), M^\sharp \rangle$
-

Fig. 4: Abstract semantics of basic commands.

5 Abstract Semantics of the Language

It now remains to specify the abstract semantic operator $\mathbb{F}^\sharp : D^\sharp \rightarrow D^\sharp$. Let $s^\sharp \in D^\sharp$. Following the specification methodology of partitioned semantic transformers [5, 4], we will define $\bar{s}^\sharp = \mathbb{F}^\sharp(s^\sharp)$ by a system of semantic equations over the variables $s@l, \bar{s}@l$ for $l \in \mathcal{L}$. We will first define the interprocedural behaviour of the program. If f is a function of P we denote by $\text{callers}(f)$ the set of command labels which correspond to a call to f , and by $\text{returns}(f)$ the set of labels which correspond to a **return** command in f . Since our language is first order this information is statically known. For the clarity of presentation, we will denote by $u ; v$ the composition $v \circ u$ of two operations u and v .

Let f be a function of P with $\text{params}(f) = \{p_1, \dots, p_n\}$. Let $\ell : x := f(x_1, \dots, x_n)$ be an element of $\text{callers}(f)$. First, we bijectively replace every variable v occurring in $\varrho(s^\sharp@l)$ with a fresh variable v' . Then, we define the abstract function call operation \mathbf{call}^\sharp as follows:

$$\mathbf{call}^\sharp(s^\sharp@l) = \langle \mathbf{bind}(\mathbf{project}_{x'_1, \dots, x'_n}(\varrho(s^\sharp@l)), \mathbf{init}), M(s^\sharp@l) \rangle$$

where

$$\begin{aligned} - \mathbf{bind} &= [\mathbf{mix} ; \mathbf{add} \{p_1 = x'_1, \dots, p_n = x'_n\} ; \mathbf{project}_{\text{vars}(f)}] \\ - \mathbf{init} &= [\mathbf{add} \{v = 0 \mid v \in \text{vars}(f) \wedge v \notin \text{params}(f)\}] (\top_{\text{vars}(f)}^\sharp) \end{aligned}$$

Intuitively, this operation amounts to transferring the numerical relationships between the arguments x_1, \dots, x_n at the caller level into the callee environment. All other local variables of the callee should be initialized to 0. The bijective renaming is necessary to avoid name clashes in the case of a recursive call. The semantic equation associated to the entry point of f can therefore be written as:

$$\bar{s}^\# @ \text{entry}(f) = \sqcup_{\text{vars}(f)} \{ \mathbf{call}^\#(s^\# @ \ell) \mid \ell \in \text{callers}(f) \}$$

Now let $\ell : x := f(x_1, \dots, x_n)$ be a function call command and $\ell' : \mathbf{return} \ y$ be a function return command in f . First, we bijectively replace every variable v occurring in $\text{vars}(f)$ with a fresh variable v' . Then, we define the abstract function return operation $\mathbf{return}_\ell^\#$ as follows:

$$\mathbf{return}_\ell^\#(s^\# @ \ell') = \langle [\mathbf{delete}_x ; \mathbf{add}\{x = y'\} ; \mathbf{project}_{\text{vars}@ \ell'}](\text{return}), M(s^\# @ \ell') \rangle$$

where $\text{return} = \mathbf{mix}(\mathbf{project}_{y'}(\varrho(s^\# @ \ell')), \varrho(s^\# @ \ell))$. Intuitively, this corresponds to transferring the information about the return value computed at the callee level back into the caller environment. As previously, the bijective renaming is necessary to avoid name clashes in the case of a recursive call.

The abstract semantics of basic assignment commands is defined in Fig. 4. The semantic equation corresponding to a program label ℓ can then be specified as follows:

$$\begin{aligned} \bar{s}^\# @ \ell = & \sqcup_{\text{vars}@ \ell} (\{ [\mathbf{cmd}]^\#(s^\# @ \ell') \mid \ell = \text{next}(\ell') \} \\ & \cup \{ \mathbf{return}_{\ell'}^\#(s^\# @ \ell'') \mid \ell' : x = f(x_1, \dots, x_n) \wedge \ell = \text{next}(\ell') \wedge \ell'' \in \text{returns}(f) \}) \end{aligned}$$

Note that, for simplicity, we have abstracted away boolean expressions. We must also add the identity transfer equation between the label ℓ of a sequential composition $\ell : \{ \ell' : \text{cmd}_1 ; \dots \}$ and the label ℓ' of its first command. This completes the definition of the semantic operator $\mathbf{F}^\#$. The soundness of our definition is ensured by the following theorem:

Theorem 2. $\mathbf{F} \circ \gamma \subseteq \gamma \circ \mathbf{F}^\#$.

The system of fixpoint equations obtained in this way can be solved by applying efficient iteration strategies [2]. In particular, it is not necessary to apply the widening operations at each control point, but only at some points which cut the cycles in the dependency graph.

Example 4. We apply the analysis to the program of Example 1 by using Karr's abstract numerical domain [17] and the results of Steensgaard's analysis described in Example 3. Then, at program point 120, we obtain the following abstract memory configuration:

$$\begin{aligned} (n_1, \text{next}, n_1) & \mapsto \{ \mathbf{s} = \mathbf{t} + 1 \} \\ (n_1, \text{value}, n_2) & \mapsto \{ \mathbf{s} = \mathbf{t} \} \\ (n_3, \square, n_2) & \mapsto \{ \mathbf{s} = 0, \mathbf{i} = \mathbf{t} \} \end{aligned}$$

together with the following abstract environment:

$$\{ \mathbf{l} = 1, \mathbf{t} = 0 \}$$

This means that the analysis has been able to discover the *exact* aliasing relationships holding at program point 120. As far as we know this is the only alias analysis which is able to give this kind of results in the presence of lists and arrays. \square

6 Conclusion

In this paper we have constructed a flow-sensitive alias analysis by Abstract Interpretation which can infer position-dependent aliasing relationships. Our work elaborates on the existing literature in this domain by providing a simpler model of the memory which can cope with recursive data structures and arrays within the same framework. This model also tends to be less costly than the ones based on an equivalence relation [11, 24]. An interesting aspect from the theoretical viewpoint is that we have designed a technique to solve a entirely symbolic problem in an purely arithmetic framework. Even the previous nonuniform alias analyses which used abstract numerical domains were heavily relying on symbolic algorithms.

We will now explore the scalability of this model to large-size programs. We are rather confident in the results since flow-sensitive analyses based on numerical domains have shown their ability to handle large programs, which is confirmed by the recent apparition of commercial tools [20]. We will also investigate more sophisticated abstractions of timestamps. In particular, we should encode the call stack more precisely in order to discover nonuniform aliasing relationships which are created recursively. We would then obtain an analysis framework which could be able to analyse recursive data structures and arrays without any restriction on the computation patterns.

Acknowledgement. The author would like to thank the anonymous referees for useful comments on a first version of this paper.

References

1. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
2. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141, 1993.
3. M. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Interprocedural pointer alias analysis. Technical report, IBM Research Report, 1997.
4. P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Conference on Principles of Programming Languages POPL'79*, pages 269–282. ACM Press, 1979.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, August 1992.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Conference on Principles of Programming Languages*. ACM Press, 1978.
9. A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL 1990*, pages 157–168, 1990.
10. A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, 1992.
11. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM Press, 1994.
12. R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of POPL 1996*, pages 1–15, 1996.
13. P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.
14. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493. Lecture Notes in Computer Science, 1991.
15. M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, 2001.
16. H.B.M Jonkers. Abstract storage structures. In De Bakker and Van Vliet, editors, *Algorithmic languages*, pages 321–343. IFIP, 1981.
17. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.
18. F. Masdupuy. Using abstract interpretation to detect array data dependencies. In *Proceedings of the International Symposium on Supercomputing*, 1991.
19. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001.
20. PolySpace Technologies. <http://www.polyspace.com>.
21. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.
22. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 1996 International Conference on Compiler Construction*, volume 1060 of *LNCS*, pages 136–150, 1996.
23. A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of the Third International Static Analysis Symposium SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer-Verlag, 1996.
24. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.