Field-Sensitive Sharing

Damiano Zanardini[a]

[a] *Technical University of Madrid, Spain*

Abstract

In static analysis of programming languages with dynamic memory, *sharing analysis* tries to infer if two variables point to data structures which are not disjoint. I.e., two variables *share* at a certain program point if there is a memory cell which can be accessed from both via two *converging paths* in the heap.

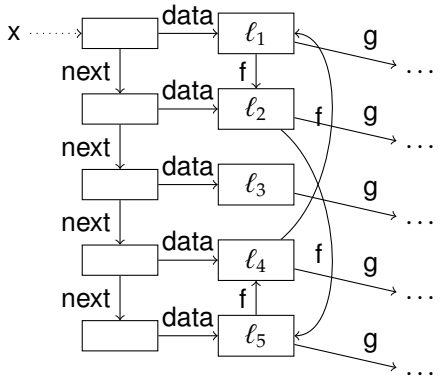
Sharing information is used as an auxiliary component in a number of static analysis techniques: to know that two variables *do not share* any memory cell allows to guarantee that any modification to the first variable has no effect on the data structure accessible from the second. On the other hand, if it cannot be guaranteed that the data structures accessible from two variables x and y are disjoint, then a loss of information occurs in that any update to x must be considered as a *possible update* of y, thus making the inference of interesting program properties much harder.

This paper introduces a novel sharing analysis which takes into account the *fields* involved in converging paths. For every two variables and every program point, a *propositional formula*, called a *path-formula*, is computed, that describes which fields may or may not be traversed by converging paths in the heap. Let x point to an object representing a phone contact, x.f point to the beginning of a single-linked list (the user's phone numbers), and x.g point to other information. On the other hand, let y point to the second element of the number list. In this case, x and y share, so that existing analyses based on sharing have to admit that an update of x.g may modify y, thus invalidating previous information about it. However, field-sensitive information like "for every two converging paths π_x and π_y from x and y, respectively, π_x does not traverse field f" allows to infer that the data structure pointed to by y will not be modified by updating x.g.

Besides improving existing static analysis techniques, the field-sensitive sharing analysis is interesting in itself, and can be formalized in the framework of *abstract interpretation* as a refinement of traditional sharing.

1. Introduction

Programming languages with dynamic memory allocation, such as Java or C++, allow creating and manipulating linked data structures in the *heap*. This feature threatens most approaches to static analysis since keeping track




```
1 while (x≠null) do {
2   y := x.data;
3   y.f := ...
4   y.g := ...
5   x := x.next;
6 }
```

Figure 1: A list storing mutually connected data, and a loop traversing it

at compile time of the possible runtime configurations of the heap is quite challenging.

One of the main sources of precision loss in static analysis of the heap is the possibility that the data structures accessible from two different variables are not disjoint: in this case, the variables are said to *share* some part of the heap, i.e., there are heap locations which can be accessed from both. This is quite a common picture in heap-manipulating object-oriented languages. The problem for static analysis techniques stems from the risk that a modification to one variable affects the data structure pointed to by the other, although this is not directly visible in the source code. *Sharing analysis* aims at proving that two variables do not share in order to rule out the possibility that an update to one breaks some interesting properties of the other. This information has been successfully used in a number of program analysis techniques and is indeed one of the fundamental components of most approaches to static analysis of heap-manipulating languages.

The present paper introduces a sharing analysis which is not tied (as many shape analyses are) to a pre-defined set of data structures or *shapes*, and can infer precise information about the fields involved in paths in the heap. It is true that most programs exhibit quite simple sharing patterns [23], so that, in many cases, there seems to be little need for a very precise sharing analysis (remember that precision is usually a major threat to scalability). However, precision *does* matter, and a significant part of the research on static analysis has always been devoted to improve the inference of interesting properties of programs (which has often become scalable in a later time). To the best of our knowledge, no existing analysis is able to infer such information, although, clearly, there exist techniques which can prove other interesting properties of the heap.

Consider a loop on the data structure drawn in Figure 1. Traditional sharing correctly infers that y shared with x after line 2. Therefore, the alteration undergone by y in lines 3 and 4 is usually considered as potentially modifying

the data structure pointed to by x. This makes it impossible to prove the *termination* of the loop since (1) there is a cycle on y (locations ℓ_1, ℓ_2, ℓ_5, ℓ_4); and (2) this cycle is accessible by x. Cycles in data structures usually make it impossible to prove the termination of loops traversing[1] the structure, because no decreasing *ranking function* can be found [5, 13, 1, 35].

However, a closer look allows to observe that any location shared by x and y is reachable (1) from x, by traversing either (1a) data; (1b) next and data; (1c) data and f; or (1d) next, data and f; and (2) from y, by traversing either (2a) no fields (i.e., the shared location can be exactly the location directly pointed to by y); or (2b) just f. A termination analyzer could use this information to infer that, by executing x:=x.next, x never accesses a cycle because field data is never traversed. Consequently, the termination of the whole loop could be proved.

The present field-sensitive sharing analysis aims at providing such kind of information about *how* variables share. This information is computed by an *abstract semantics* that manipulates *propositional formulæ* representing, for a pair of variables, the fields which have to be traversed by heap paths reaching the same location (*converging paths*). For example, suppose that next, data, f and g are the only fields in the program: the above sharing information between x and y (i.e., the pair (x, y)) can be represented as

$$\grave{\mathsf{d}}\mathsf{ata}_{\searrow} \wedge \neg \grave{\mathsf{g}}_{\searrow} \wedge \neg \acute{\mathsf{d}}\mathsf{ata}_{\swarrow} \wedge \neg \acute{\mathsf{n}}\mathsf{ext}_{\swarrow} \wedge \neg \acute{\mathsf{g}}_{\swarrow}$$

meaning that (1) any path from the first variable x (downward arrows from left to right correspond to the first variable in the pair) to a shared location has to traverse data (as indicated by $\grave{\mathsf{d}}\mathsf{ata}_{\searrow}$) but not g (as specified by $\neg \grave{\mathsf{g}}_{\searrow}$), and, implicitly, may traverse either next or f, or both; and (2) any path from y (corresponding to arrows from right to left) can only possibly traverse f, the traversal of the other fields being explicitly forbidden ($\neg \acute{\mathsf{d}}\mathsf{ata}_{\swarrow} \wedge \neg \acute{\mathsf{n}}\mathsf{ext}_{\swarrow} \wedge \neg \acute{\mathsf{g}}_{\swarrow}$).

Following the well-known theory of *abstract interpretation* [14], an *abstract domain* representing field-sensitive sharing is defined (Section 3), and a sound *abstract semantics* is built on it (Section 4), which computes the desired information. A prototypical implementation of the abstract semantics is presented in Section 5.

Main contributions.

As discussed in Section 1.1, a closely-related work [38] introduces field-sensitive reachability and cyclicity analysis. The present paper builds on that work and adds the following contributions:

- The definition of an abstract domain capturing field-sensitive sharing information in the form of propositional formulæ attached to pairs of variables at a given program point; reachability is a special case of sharing, so that this paper generalizes the previous one.

[1] To "traverse" a field, a path in the heap or a whole data structure has to be understood as field dereference: the path from the location pointed to by v to the one pointed to by $v.f$ traverses φ.

- Unlike the existing reachability and cyclicity analysis, a new representation of propositional formulas is used: instead of being identified by the set of their models, formulas are built and manipulated syntactically by using standard logical operators. Moreover, instead of having one path-formula for every pair of variables, a single formula is used to represent sharing information about *all* variables at a given program point; this approach is amenable to be implemented efficiently.

- A sound abstract semantics is built on the abstract domain with the new representation. Soundness proofs are provided for the most complex rules: field access, field update and method call.

- The approach has been implemented. Unlike previous work, a more complete, interprocedural implementation is provided.

1.1. Related work

This section starts with a short description of related analyses of heap-manipulating program; its goal is to introduce concepts more than discuss related work. On the other hand, the second part explores the relation of the present work with shape analysis: most importantly, it argues that there are two main aspects for which our analysis is different from related shape analyses. Finally, the third part goes back to the related ideas of the first part, and reviews related work: our analysis infers information which is not inferred by related approaches.

Sharing and similar approaches. The present paper is related to research on *pointer analysis* [22], which considers properties of the heap and builds *static analyses* to enforce them. Apart from sharing, *aliasing*, *points-to*, *shape*, *reachability* and *cyclicity* are the most related properties for which pointer analyses can be found in the literature.

Sharing analysis [33] determines if two variables v_1 and v_2 can reach a common location in the heap, i.e., if the portions of the heap which are reachable from v_1 and v_2 are not disjoint. The usual approach to sharing analysis is *possible pair-sharing*[2]: for every pair of variables v_1 and v_2, the analysis tries to infer if it is possible that two paths in the heap, each one starting from one of the variables, *converge* to the same memory location. In this case, sharing pairs of variables are over-approximated, that is, two variables are supposed to (possibly) share unless the contrary can be proved. On the other hand, a *definite sharing* analysis can also be defined, where variable pairs are under-approximated: two variables are supposed not to share unless it can be proved that they do.

A well-known technique in pointer analysis, *aliasing analysis* [22], investigates the program variables which might point to the same heap location at

[2]Set-sharing is a closely related notion which, in logic programming languages, can possibly contain information which is not dealt with by pair-sharing [12]. However, it does not seem to be interesting in itself in the imperative paradigm.

runtime. Aliasing between two variables implies that they also share, but the other direction is normally false. In particular, the presented sharing analysis also deals with aliasing since it is possible to describe aliasing by two empty paths (traversing no fields) converging to the same location. *Points-to analysis* computes (an approximation of) the set of objects which might be referred to by a pointer variable.

Cyclicity analysis [19, 30, 17, 18, 27] tries to infer if it is possible to access a *cycle* from a variable, and is a crucial component of termination analyses of heap-manipulating programs [10]. Apart from sharing, one of the basic components of a cyclicity analysis is *reachability analysis* [26, 19, 17, 18, 27], which computes pairs of variables such that there can be a path in the heap from one to (the location pointed to by) the other. Sharing and reachability are closely related because a variable only reaches another one if it also shares with it. Therefore, sharing has to be regarded as a generalization of both aliasing and reachability.

Shape analysis. Research on *shape analysis* [37] reasons about heap-manipulating programs in order to prove program properties. In most cases, *safety* properties are dealt with [4, 31, 29]. On the other hand, work on *liveness* [28, 3, 5, 13, 11] uses techniques based on *model checking* [24], *predicate abstraction* [20], *separation logic* [28] or *cyclic proofs* [11] in order to prove properties of programs manipulating the heap. Such techniques are quite precise, at the cost of (i) limiting the shape of the data structures which can be analyzed; (ii) simplifying the programming language to be dealt with; or (iii) reducing scalability.

Unlike most of these approaches, the present analysis infers sharing information *as it is* (i.e., the focus is simply on how two variables share), without being tailored to a specific program property to be proven. In contrast, typical shape analyses are designed with specific properties in mind (for example, the absence of dangling pointers): this allows to prove precise results by a suitable and *ad hoc* representation of data structures, but also limits the flexibility of the approach.

Moreover, the presented sharing analysis does not consider a limited set of shapes (list, tree, DAG, etc.): instead, the information is gathered without relying on an *a priori* description (e.g., an inductive definition) of how a data structure is supposed to be. This is a major difference, and one of the most important motivations for this analysis.

More concretely, some existing works in shape analysis will be shortly reviewed in order to determine some similarities and difference with respect to the present approach. The $\mathcal{LSL}$ Storeless Semantics [29] describes each location in the heap by the set of paths reaching it; paths include the name of fields, so that their approach is similar to ours in this sense. However, there are a number of important differences:

- a crucial one: $\mathcal{LSL}$ is a *concrete* semantics, not an abstract one;
- its abstract counterpart, also described in the paper, uses 3-Valued logical structures to represent sets of memory states; however, unlike our approach, it works on objects with a single field; on the other hand, one

of the core ideas of the present technique is to use propositional formulas in order to describe the different fields traversed by paths;

- as mentioned before, their abstract description of the heap is designed in order to prove some specific safety property of an algorithm on a specific data structure (a single-linked list); on the other hand, our sharing analysis in independent of the data structure and the properties to be eventually studied.

Similarly, *Regular Model Checking* and *Abstract Regular Model Checking* [9, 7] deal with verifying properties of algorithms on Dynamic Linked Data Structures. Again, only pre-defined data structures where objects have a unique field (selector) are taken into account (a possibly cyclic linked list), and properties of specific algorithms (e.g., list reversal) are proved. On the other hand, *Abstract Regular Tree Model Checking* [8] does deal with multiple fields, and can prove properties of structures such as double-linkd lists and non-trivial trees; anyway, the kind of data structures is known in advance. Again, the focus is on specific safety properties like the consistency of pointer manipulation or the preservation of certain invariants. In this work, the absence of sharing can be encoded as an invariant, and shape analysis can prove that such an invariant is preserved; this is quite a distant approach from the present one, which describes sharing patterns without starting from any prior information; moreover, their sharing is field-insensitive.

Field-sensitive related approaches. A simple sharing analysis [33] has some limitations in the way it can be used as a component of a cyclicity analysis: updating a field of an object o with another object o' sharing with o automatically makes the analysis believe that a cycle has been created; this threatens the possibility to prove properties like termination in a number of situations. Therefore, there have been several proposals in order to refine the information about the heap and be able to prove that, in certain cases, (a) a cycle is not created; or (b) the characteristics of a cycle still allow to prove termination.

In the first direction, reachability has been used [26, 19, 17, 18, 27] to replace sharing in order to prove more precise results: continuing the previous example, the analysis only believes that a cycle has been created if o' was reaching o, not only sharing with it. Since reachability impies sharing, this approach can be seen as a refinement of the simplest sharing analyses.

In direction (b), some of the properties discussed in the first part of this section have been recently refined by taking into account the fields involved in paths. Scapin and Spoto [32] are able to infer that certain cycles in a data structure do *not* involve a field, so that traversals of the structure on that field are guaranteed to terminate. Brockschmidt et al. [10] can guarantee the termination of a one-direction traversal of a double-linked list since they can prove that every cycle in the list must involve both next and prev.

A recent paper [38] builds on these works to define a field-sensitive reachability and cyclicity analysis which is even more precise. Propositional formulæ represent the fields which have or have not to be present in paths in the

heap. This approach can represent information like "every path must traverse field f OR field g", which could not be inferred before. The present paper follows the last approach and brings it to field-sensitive sharing. The propositional formula used in field-sensitive sharing is also able to express aliasing and reachability information, and is more complex than the one used in inferring field-sensitive reachability because two paths have to be considered instead of one. More detailed contributions of this work have been already discussed at the end of Section 1.

2. A simple object-oriented language

This section defines the syntax and the denotational semantics of a simple, Java-like object-oriented language. Name for classes, methods, fields and variables are taken from a set $\mathcal{X}$ of valid *identifiers*. A *program* is a set of classes $\mathcal{K} \subseteq \mathcal{X}$ partially ordered by the *subclass* relation $\prec$. A *class declaration* takes the form "**class** κ_1 [**extends** κ_2] { $t_1\ \varphi_1$;… $t_n\ \varphi_n$; $M_1 \ldots M_k$}" where each "$t_i\ \varphi_i$" declares the field φ_i to have type $t_i \in \mathcal{K} \cup \{\textbf{int}\}$, and each M_j is a method definition. The optional statement "**extends** κ_2" declares κ_1 to be a subclass of κ_2. *Method definitions* take the form "$t\ mth(t_1\ w_1,\ldots,t_n\ w_n)\ \{t_{n+1}\ w_{n+1};\ldots t_{n+p}\ w_{n+p};\ com\}$" where: mth is the method name; $t \in \mathcal{K} \cup \{\textbf{int}\}$ is the type of the return value; $w_1,\ldots,w_n$ are formal parameters; $w_{n+1},\ldots,w_{n+p}$ are local variables; t_{n+k} is the *declared type* of w_{n+k}, hereafter denoted by $\delta(w_{n+k})$; and the command com follows this grammar:

$$
\begin{array}{rl}
exp ::= & n \mid \textbf{null} \mid v \mid v.\varphi \mid exp_1 \oplus exp_2 \mid \textbf{new}\ \kappa \mid v.mth(\bar{v}) \\
com ::= & \textbf{skip} \mid v{:=}exp \mid v.\varphi{:=}exp \mid com_1\,;com_2 \mid \\
& \textbf{if}\ exp\ \textbf{then}\ com_1\ [\textbf{else}\ com_2] \mid \textbf{while}\ exp\ \textbf{do}\ com \mid \textbf{return}\ exp
\end{array}
$$

where $v, mth, \varphi \in \mathcal{X}$; $\bar{v} \in \mathcal{X}^*$; $n \in \mathbb{Z}$; $\kappa \in \mathcal{K}$; and $\oplus$ is a binary operator on **int**. For simplicity, *conditions* in **if** and **while** statements are assumed not to have side effects. A *method signature* $\kappa.mth(t_1,\ldots,t_n){:}t$ refers to a method mth defined in class κ, taking n parameters of type $t_1,\ldots,t_n \in \mathcal{K} \cup \{\textbf{int}\}$, and returning a value of type t. Given a signature mth, let mth^b be its body; mth^i be its input variables $\{this, w_1,\ldots,w_n\}$, where $this$ refers to the object receiving the call; mth^l be its local variables $\{w_{n+1},\ldots,w_{n+m}\}$; and $\mathsf{mth}^s = \mathsf{mth}^i \cup \mathsf{mth}^l$. Command sequences will be wrapped by {}. Given a program, $\mathcal{F}$ is the set of fields declared in it[3].

A *type environment* τ is a partial map from $\mathcal{X}$ to $\mathcal{K} \cup \{\textbf{int}\}$ which associates types to variables at a given program point. Abusing notation, when it is clear from the context, type environments will be confused with sets of variables when types are not important; i.e., $v \in \tau$ will stand for $v \in \mathrm{dom}(\tau)$. A *state* over τ is a pair consisting of a frame and a heap. A *heap* μ is a partial mapping from an infinite and totally ordered set $\mathcal{L}$ of memory locations to objects; $\mu(\ell)$ is the object bound to $\ell \in \mathcal{L}$ in the heap μ. An *object* $o \in \mathcal{O}$ is a pair consisting of a class

[3]For simplicity, **int** fields will be often ignored since they have no impact on sharing.

$$
\begin{aligned}
E^\iota_\tau[\![n]\!](\sigma) &= \left\langle \sigma^f[\rho \mapsto n], \sigma^h \right\rangle \\
E^\iota_\tau[\![\textbf{null}]\!](\sigma) &= \left\langle \sigma^f[\rho \mapsto \mathit{null}], \sigma^h \right\rangle \\
E^\iota_\tau[\![\textbf{new}\ \kappa]\!](\sigma) &= \left\langle \sigma^f[\rho \mapsto \ell], \sigma^h[\ell \mapsto \mathit{newobj}(\kappa)] \right\rangle \text{ where } \ell \notin \mathrm{dom}(\sigma^h) \\
E^\iota_\tau[\![v]\!](\sigma) &= \left\langle \sigma^f[\rho \mapsto \sigma^f(v)], \sigma^h \right\rangle \\
E^\iota_\tau[\![v.\varphi]\!](\sigma) &= \left\langle \sigma^f[\rho \mapsto \sigma^h(\sigma^f(v)).\varphi], \sigma^h \right\rangle \\
E^\iota_\tau[\![\mathit{exp}_1 \oplus \mathit{exp}_2]\!](\sigma) &= \left\langle \sigma^f\left[\rho \mapsto \sigma^f_1(\rho) \oplus \sigma^f_2(\rho)\right], \sigma^h_2 \right\rangle \text{ where} \\
&\qquad \sigma_1 = E^\iota_\tau[\![\mathit{exp}_1]\!](\sigma) \text{ and } \sigma_2 = E^\iota_\tau[\![\mathit{exp}_2]\!]\left(\left\langle \sigma^f, \sigma^h_1 \right\rangle\right) \\
E^\iota_\tau[\![v_0.\mathit{mth}(v_1,\ldots,v_n)]\!](\sigma) &= \left\langle \sigma^f\left[\rho \mapsto \sigma^f_2(\mathit{out})\right], \sigma^h_2 \right\rangle \text{ where } \sigma_2 = \iota(\mathsf{mth})(\sigma_1) \text{ s.t. } \sigma_1 \text{ is} \\
&\qquad \sigma^h_1 = \sigma^h; \sigma^f_1(\mathit{this}) = \sigma^f(v_0); \forall 1 \le i \le n.\ \sigma^f_1(w_i) = \sigma^f(v_i); \\
&\qquad \text{and } \mathsf{mth} = \mathit{lookup}(\sigma, v_0.\mathit{mth}(v_1,\ldots,v_n)); \\
\hline
C^\iota_\tau[\![\textbf{skip}]\!](\sigma) &= \sigma \\
C^\iota_\tau[\![v{:=}\mathit{exp}]\!](\sigma) &= \left\langle \sigma^f\left[v \mapsto \sigma^f_e(\rho)\right], \sigma^h_e \right\rangle \\
C^\iota_\tau[\![v.\varphi{:=}\mathit{exp}]\!](\sigma) &= \left\langle \sigma^f, \sigma^h\left[\ell.\varphi \mapsto \sigma^f_e(\rho)\right] \right\rangle \text{ where } \ell = \sigma^f(v) \\
C^\iota_\tau\left[\!\!\left[\begin{array}{ll}\textbf{if } \mathit{exp} & \textbf{then } \mathit{com}_1 \\ & \textbf{else } \mathit{com}_2\end{array}\right]\!\!\right](\sigma) &= \text{if } \sigma^f_e(\rho) \neq 0 \text{ then } C^\iota_\tau[\![\mathit{com}_1]\!](\sigma) \text{ else } C^\iota_\tau[\![\mathit{com}_2]\!](\sigma) \\
C^\iota_\tau[\![\textbf{while } \mathit{exp} \textbf{ do } \mathit{com}]\!](\sigma) &= \delta(\sigma) \text{ where } \delta \text{ is the least fixpoint of} \\
&\qquad \lambda w.\lambda\sigma.\ \text{if } \sigma^f_e(\rho) \neq 0 \text{ then } w(C^\iota_\tau[\![\mathit{com}]\!](\sigma)) \text{ else } \sigma \\
C^\iota_\tau[\![\textbf{return } \mathit{exp}]\!](\sigma) &= \left\langle \sigma^f\left[\mathit{out} \mapsto \sigma^f_e(\rho)\right], \sigma^h_e \right\rangle \\
C^\iota_\tau[\![\mathit{com}_1; \mathit{com}_2]\!](\sigma) &= C^\iota_\tau[\![\mathit{com}_2]\!](C^\iota_\tau[\![\mathit{com}_1]\!](\sigma))
\end{aligned}
$$

Figure 2: Denotations for expressions and commands. The state σ_e is $E^\iota_\tau[\![\mathit{exp}]\!](\sigma)$.

tag $o.\mathsf{tag} \in \mathcal{K}$, and a frame $o.\mathsf{frm}$ which maps its fields to $\mathcal{V} = \mathbb{Z} \cup \mathcal{L} \cup \{\mathit{null}\}$. It is assumed that no two fields $\kappa.\varphi$ and $\kappa'.\varphi$ with the same name can be declared in a program, so that φ will be usually a shorthand for $\kappa.\varphi$; this is not a significant restriction w.r.t. Java since the actual field to which a Java expression $v.\varphi$ may refer to is known statically. The shorthand $o.\varphi$ is used instead of $o.\mathsf{frm}(\varphi)$. A *frame* ϕ maps variables in $\mathrm{dom}(\tau)$ to $\mathcal{V}$. For $v \in \mathrm{dom}(\tau)$, $\phi(v)$ refers to the value of v, and $\phi[v \mapsto \mathit{val}]$ is the frame where v has been set to val, or defined to be val if $v \notin \mathrm{dom}(\phi)$. The set of states over τ is

$$
\Sigma_\tau = \left\{ \left\langle \phi, \mu \right\rangle \;\middle|\; \begin{array}{ll} 1. & \phi \text{ is a frame over } \tau, \mu \text{ is a heap, and both are well-typed} \\ 2. & \mathrm{rng}(\phi) \cap \mathcal{L} \subseteq \mathrm{dom}(\mu) \\ 3. & \forall \ell \in \mathrm{dom}(\mu).\ \mathrm{rng}(\mu(\ell).\mathsf{frm}) \cap \mathcal{L} \subseteq \mathrm{dom}(\mu) \end{array} \right\}
$$

Given $\sigma \in \Sigma_\tau$, σ^f and σ^h refer to its frame and its heap, respectively. The lattice $\mathcal{I}^\tau_\flat = \langle \wp(\Sigma_\tau), \top, \bot, \cap, \cup \rangle$ defines the *concrete domain*, where $\top{=}\Sigma_\tau$ and $\bot{=}\emptyset$.

A *denotation* δ over type environments τ_1 and τ_2 is a partial map from Σ_{τ_1} to Σ_{τ_2}: it describes how the state changes when some code is executed. The set of denotations from τ_1 to τ_2 is $\Delta(\tau_1,\tau_2)$. An *interpretation* ι is a special denotation giving a meaning to methods in terms of input and output variables: it maps methods to denotations, s. t. $\iota(\mathsf{mth}) \in \Delta(\mathsf{mth}^i,\{\mathit{out}\})$ for each mth. The variable out is a special variable denoting the return value. Γ is the set of all interpretations.

Denotations for expressions and commands are depicted in Figure 2. $E^\iota_\tau[\![\mathit{exp}]\!]$

maps states from Σ_τ to states from $\Sigma_{\tau\cup\{\rho\}}$, where ρ is a special variable for storing the value of *exp*. A command denotation $C^\iota_\tau[\![com]\!]$ maps states to states, in presence of $\iota \in \Gamma$. The function *newobj*(κ) creates a new instance of κ with **int** fields initialized to 0 and reference fields initialized to *null*, while *newloc*(σ^h) returns the first *free* location, i.e., the first $\ell \notin \mathrm{dom}(\sigma^h)$ according to a total ordering on locations. The function *lookup* resolves the method call based on the runtime type of the object, and returns the signature to be invoked. The *concrete denotational semantics* of a program is defined as the *least fixpoint* (*lfp*) of the following transformer of interpretations [6].

Definition 2.1. *The denotational semantics of a program P is the* lfp *of*

$$T_P(\iota) = \left\{\mathsf{mth} \mapsto \lambda\sigma{\in}\Sigma_{\mathsf{mth}^i}.\exists\tau\backslash out.C^\iota_{\mathsf{mth}^s\cup\{out\}}[\![\mathsf{mth}^b]\!](ext(\sigma,\mathsf{mth}))\right\}_{\mathsf{mth}\in P}$$

where $ext(\sigma,\mathsf{mth}) = \left\langle \sigma^f[\forall v \in \mathsf{mth}^l \cup \{out\}.v \mapsto 0/null], \sigma^h\right\rangle$.

The denotation for a method signature $\mathsf{mth} \in P$ is computed by T_P as follows: it (1) extends (using $ext(\sigma,\mathsf{mth})$) the input state $\sigma \in \Sigma_{\mathsf{mth}^i}$ such that local variables are set to 0 or *null*; (2) computes the denotation of the code of mth, using $C^\iota_{\mathsf{mth}^s\cup\{out\}}[\![\_]\!]$; and (3) restricts the resulting denotation to *out*, using $\exists\tau\backslash out$.

3. The abstract domain

This section formalizes the field-sensitive sharing analysis by means of Abstract Interpretation [14], relying on the notion of *abstract domain*.

3.1. Background in Logic

A *boolean function* is a function $f : Bool^n \mapsto Bool$ with $n \geq 0$, and can be represented as a *propositional formula* over a set X with cardinality n. In this paper, boolean functions and propositional formulæ will be used interchangeably. X will be the set $\mathcal{P} = \{\searrow\!\varphi, \swarrow\!\varphi \mid \varphi \in \mathcal{F}\}$ of propositions $\searrow\!\varphi$ and $\swarrow\!\varphi$ corresponding to program fields. Such propositions are called *f-propositions*. For every field φ there are two f-propositions, depending on which variable of the sharing pair is considered (see below): the *left* one, $\searrow\!\varphi$, and the *right* one, $\swarrow\!\varphi$. In the following, $\updownarrow\!\phi$ will denote an f-proposition when there is no need to refer to a specific variable. Propositional formulæ over $\mathcal{P}$ are called *path-formulæ*. A *truth assignment* $\omega \subseteq \mathcal{P}$ is a *model* of a path-formula F if F evaluates to *true* under ω; set notation for truth assignments means that $p \in \mathcal{P}$ is assigned to *true* in ω iff $p \in \omega$. The set of models of F is denoted by $mdls_{\mathcal{P}}(F)$.

For a set ω, the path-formula $\langle\omega\rangle$ is defined as $\bigwedge\{\updownarrow\!\phi \mid \updownarrow\!\phi{\in}\omega\} \wedge \bigwedge\{\neg\updownarrow\!\phi \mid \updownarrow\!\phi{\in}\mathcal{P}\backslash\omega\}$, and represents the formula whose *only* model is ω. An important special case is $\langle\emptyset\rangle = \bigwedge\{\neg\updownarrow\!\phi \mid \updownarrow\!\phi{\in}\mathcal{P}\}$; moreover, $\left\langle\updownarrow\!\phi_1,..\updownarrow\!\phi_k\right\rangle$ will be a shorthand for $\left\langle\{\updownarrow\!\phi_1..\updownarrow\!\phi_k\}\right\rangle$. Finally, $\searrow\!\omega$ and $\swarrow\!\omega$ denote, resp., the *left* and *right* projection of ω:

$$\searrow\!\omega = \left\{\searrow\!\varphi \;\middle|\; \searrow\!\varphi \in \omega\right\} \qquad\qquad \swarrow\!\omega = \left\{\swarrow\!\varphi \;\middle|\; \swarrow\!\varphi \in \omega\right\}$$

As usual, *true* stands for a tautology, and *false* stands for a contradiction. Ordering on path-formulæ is logical implication: $F' \leq F''$ iff $F' \Rightarrow F''$ is valid.

3.2. Paths, cycles, and fields

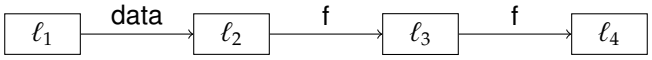
The abstract domain used in this paper is based on *reachable heap locations*, i.e., the part of the heap which can be reached from a given location (or the variable pointing to it). Given a heap μ, a *path* π from $\ell' \in \mathrm{dom}(\mu)$ to $\ell'' \in \mathrm{dom}(\mu)$ is a sequence $\langle \ell_0, .., \ell_k \rangle$ of locations s. t. (1) $k \geq 0$; (2) $\ell_0 = \ell'$; (3) $\ell_k = \ell''$; and (4) for every $0 \leq i \leq k-1$, it holds that $\ell_{i+1} \in \mathrm{rng}(\mu(\ell_i).\mathsf{frm})$, i.e., ℓ_{i+1} is the location bound to a field of the object to which ℓ_i is bound. The *length* of a path $\langle \ell_0, .., \ell_k \rangle$ is k; the *empty* path π_ε is the one with length 0. A *cycle* is a path from ℓ to ℓ itself. Given $\pi_1 = \langle \ell_0, .., \ell_k \rangle$ and $\pi_2 = \langle \ell_k, .., \ell_m \rangle$, the *concatenation* $\pi_1 \cdot \pi_2$ is $\langle \ell_1, .., \ell_k, .., \ell_m \rangle$.

Definition 3.1 ([30]). *The set of* reachable locations *from ℓ is $R(\mu, \ell) = \cup \{R^i(\mu, \ell) \mid i \geq 0\}$, where $R^0(\mu, \ell) = \{\ell\}$, and $R^{i+1}(\mu, \ell)$ is $\cup\{\mathrm{rng}(\mu(\ell').\mathsf{frm}) \cap \mathcal{L} \mid \ell' \in R^i(\mu, \ell)\}$.*

The rest of this section is developed in the context of a type environment τ, which will be often left implicit. *Fields* or *field identifiers* will be considered when collecting information about paths; to this end, the domain relies on the notion of *field-reachable heap locations*, i.e., the part of the heap which can be reached from a location by traversing (dereferencing) certain fields.

Definition 3.2 (field traversal). *A path π is said to* traverse *$\varphi \in \mathcal{F}$ in σ if (1) it is a path in σ^h; (2) $\pi = \langle \ell_0, .., \ell_i, .., \ell_{i+1}, .., \ell_k \rangle$ with $k > i \geq 0$; (3) an object o of class $\kappa' \preceq \kappa$ is stored in ℓ_i; and (4) $o.\varphi$ points to the location ℓ_{i+1}, i.e., $o.\mathsf{frm}(\varphi) = \ell_{i+1}$.*

Example 3.3 (field traversal). *The path depicted below traverses fields* data *and* f.



Unlike previous work [38], p-satisfaction will be defined on pairs of paths.

Definition 3.4 (p-satisfaction on pairs). *A pair of paths (π_1, π_2) is said to* p-satisfy *a path-formula F if the truth assignment*

$$\left\{\varphi_{\searrow} \mid \pi_1 \textit{ traverses } \varphi\right\} \cup \left\{\varphi_{\swarrow} \mid \pi_2 \textit{ traverses } \varphi\right\}$$

is a model of F. π_1 and π_2 are the left *and* right element *(or* path*) of the pair.*

The intuition behind this definition is that π_1 and π_2 have to be considered as the paths starting from two variables and reaching the same location. The meaning of the ordering on path-formulæ is now straightforward: for every pair (π_1, π_2), if $F' \leq F''$ and (π_1, π_2) p-satisfies F', then it also p-satisfies F''.

Example 3.5. *Consider, again, the path of Example 3.3, and let it be denoted as π. The pair (π, π) p-satisfies* $\mathsf{data}_{\searrow} \wedge \mathsf{f}_{\searrow} \wedge \mathsf{data}_{\swarrow} \wedge \mathsf{f}_{\swarrow}$ *since a model of the path-formula is the truth assignment* $\{\mathsf{data}_{\searrow}, \mathsf{f}_{\searrow}, \mathsf{data}_{\swarrow}, \mathsf{f}_{\swarrow}\}$*, and both the left and the right path (which are actually the same path) traverse both* data *and* f.

A truth assignment $\omega \subseteq \mathcal{P}$ is said to be *viable* if there exists some pair (π_1, π_2) in some state σ which p-satisfies $\langle\omega\rangle$. Moreover, two path-formulæ F and G are *equivalent* unless there is a pair of paths in some state which p-satisfies one and only one of them. Previous work [38] established that viability of truth assignments and equivalence of path-formulæ are decidable properties in a slightly different framework where there is only one (instead of two) proposition for every program field; these results are still valid here. In the following, $\mathcal{PF}_{\equiv}$ will be $\mathcal{PF}$ equipped with the natural equivalence relation just mentioned. The operators on path-formulæ introduced in the rest of this paper only consider viable truth assignments; this is sensible since viability is decidable. Inviable truth assignments are those which are not compatible with the class hierarchy of the program: for example, it can be the case that no path can traverse both `C.f` and `C.g` because of the following class declarations:

```
class C { D f; }        class C { D g; }        class D {}
```

It is easy to see that any path traversing either `f` or `g` is forced to "end" immediately, so that it is impossible to traverse both.

3.3. The Field-Sharing domain

Definition 3.6 (field-sharing). *Two variables v_1 and v_2 are said to* share *in σ if there exist two paths π_1 and π_2 in the heap σ^h such that π_1 starts from $\sigma^f(v_1)$, π_2 starts from $\sigma^f(v_2)$, and both end in the same location (i.e., they* converge*). Moreover, given a path-formula F, v_1 and v_2 are said to F-*share *in σ if, for* every *π_1 and π_2 satisfying the conditions already mentioned, (π_1, π_2) p-satisfies F. This definition implies that any two variables v_1 and v_2 false-share iff they do not share in the traditional sense.*

In the following, two *converging paths* from (v_1, v_2) will be two paths starting from $\sigma^f(v_1)$ and $\sigma^f(v_2)$, resp., and reaching the same ℓ. Moreover, v_1 will be said to *reach* v_2 if there is a path π from $\sigma^f(v_1)$ to $\sigma^f(v_2)$. Reachability is captured by sharing since $\sigma^f(v_2)$ happens to be the common location, so that π and the (empty) path from $\sigma^f(v_2)$ to itself are converging. Notation will be occasionally abused by denoting the path from $\sigma^f(v_1)$ to $\sigma^f(v_2)$ as "the path from v_1 to v_2".

An extension of the equivalence relation on path-formulæ is needed here: $\mathcal{PF}'_{\equiv}$ is a function which takes two variables v_1 and v_2, and returns the set $\mathcal{PF}$ equipped by the following equivalence relation $\equiv^{v_1,v_2}$: $F \equiv^{v_1,v_2} G$ unless there are two paths π_1 and π_2 converging from (v_1, v_2) such that the pair (π_1, π_2) p-satisfies one and only one between F and G. The only difference w.r.t. the original $\equiv$ is that the paths must start from v_1 and v_2. The following definition shows the lattice of abstract values representing sharing between variables. In the following, functions are often represented by *λ-notation*, and τ is omitted.

Definition 3.7. *The* field-sharing abstract domain *is the complete lattice*

$$\bar{\mathcal{I}}_s = \left\langle \bar{\mathcal{S}},\ \bar{\sqsubseteq}_s,\ \bar{\bot}_s,\ \bar{\top}_s,\ \bar{\sqcap}_s,\ \bar{\sqcup}_s \right\rangle$$

- $\bar{\mathcal{S}}$ *is the set of functions* I_s *whose domain is* $\mathcal{X} \times \mathcal{X}$*, and that return an element of* $\mathcal{PF}'(v_1, v_2) = \mathcal{PF}_{\equiv^{v_1,v_2}}$ *for a pair of arguments* (v_1, v_2)*;*
- $I'_s \bar{\sqsubseteq}_s I''_s$ *iff, for every* v_1 *and* v_2*,* $I'_s(v_1, v_2) \leq I''_s(v_1, v_2)$ *(remember that* $\leq$ *is logical implication);*
- $\bar{\bot}_s = \lambda(v, w).$ *false and* $\bar{\top}_s = \lambda(v, w)$ *true;*
- $I'_s \bar{\sqcap}_s I''_s = \lambda(v, w).\ I_s(v, w) \wedge I''_s(v, w)$*;*
- $I'_s \bar{\sqcup}_s I''_s = \lambda(v, w).\ I'_s(v, w) \vee I''_s(v, w)$*.*

The meaning of an abstract value I_s is the following: it represents all the states where, for every v_1 and v_2, and for all two converging paths π_1 and π_2 starting from v_1 and v_2, respectively, (π_1, π_2) p-satisfies $I_s(v_1, v_2)$.

Note that $F \not\equiv$ *false* does not mean that there is actually some sharing between v_1 and v_2 in a concrete state: this a "possible" analysis, so that non-sharing is always a possibility. On the other hand, $\bar{I}_r(v, w) =$ *false* excludes sharing since no pair of paths p-satisfies *false*. The top element $\bar{\top}_s$ represents Σ_τ. The bottom element $\bar{\bot}_s$ models the (non-empty) set of all states where all reference variables are **null**; this is so since there can be no sharing at all in states modeled by $\bar{\bot}_s$, not even self-sharing (i.e., sharing between a variable and itself), and the absence of self-sharing on a variable is equivalent to its nullity. In fact, $I_s(v, v) \geq \langle\emptyset\rangle$ (recall that $\langle\emptyset\rangle$ is $\bigwedge\{\neg\not\!\!\downarrow \mid \not\!\!\downarrow \in \mathcal{P}\}$) whenever v is not **null**. This means that there are always two converging empty paths (actually, they are the same path starting from $\sigma^f(v)$) starting from a non-null variable and reaching a common location (actually, that same location $\sigma^f(v)$), so that the pair consisting of such two paths p-satisfies $\langle\emptyset\rangle$. In general, the treatment of empty paths makes this abstract domain able to represent *aliasing* [22].

Definition 3.8 (abstraction and concretization). *The* abstraction *and* concretization *functions between* $\bar{\mathcal{I}}_s$ *and the concrete domain* $\mathcal{I}^\tau_\flat$ *are:*

$$\begin{array}{lcl} \bar{\alpha}'_s(\sigma) & = & \lambda(v_1, v_2).\ \bigwedge\ \{\, F \mid v_1 \text{ and } v_2 \text{ } F\text{-share in } \sigma \,\} \\ \bar{\alpha}_s(I_\flat) & = & \lambda(v_1, v_2).\ \bigvee\ \{\, \bar{\alpha}'_s(\sigma)(v_1, v_2) \mid \sigma \in I_\flat \,\} \\ \bar{\gamma}_s(I_s) & = & \{\, \sigma \in \Sigma_\tau \mid\ \forall v_1, v_2 \in \tau.\ \exists F \leq I_s(v_1, v_2).\ v_1 \text{ and } v_2 \text{ } F\text{-share in } \sigma \,\} \end{array}$$

Notation will be abused by writing $\bar{\alpha}_s(\sigma)$ *instead of* $\bar{\alpha}_s(\{\sigma\})$ *when the set is a singleton.*

$\bar{\alpha}_s$ is computed as follows: for a state σ, the conjunction of all F such that v_1 and v_2 F-share is the *strongest condition* which is p-satisfied by all sharing paths between v_1 and v_2 in σ (recall that the p-satisfaction must hold for *all* pairs of paths). Then, all strongest conditions are combined by disjunction because all states in $I_\flat$ must be taken into account. On the other hand, $\bar{\gamma}_s$ is the adjoint function which guarantees the Galois insertion between $\bar{\mathcal{I}}_s$ and $\mathcal{I}^\tau_\flat$.

Lemma 3.9 (insertion). $\bar{\alpha}_s$ *and* $\bar{\gamma}_s$ *identify a* Galois insertion *between* $\bar{\mathcal{I}}_s$ *and* $\mathcal{I}^\tau_\flat$*.*

It is straightforward to observe that $\bar{\mathcal{I}}_s$ is a refinement of the traditional domain for pair-sharing Sh_τ [33]. The abstraction and concretization functions α_0 and γ_0 between $\bar{\mathcal{I}}_s$ and Sh_τ simply rule out any field-sensitive information.

$$\begin{array}{lcl} \alpha_0(I_s) & = & \{(v_1, v_2) \mid I_s(v_1, v_2) \not\equiv \mathit{false}\} \\ \gamma_0(sh) & = & I_s \text{ such that } I_s(v_1, v_2) = \mathit{true} \text{ for every } (v_1, v_2) \in sh \end{array}$$

4. The field-sensitive abstract semantics

This section introduces an abstract semantics for field-sensitive sharing based on $\bar{\mathcal{I}}_s$. An *abstract denotation* ξ from τ_1 to τ_2 is a partial map from $\bar{\mathcal{I}}_s^{\tau_1}$ to $\bar{\mathcal{I}}_s^{\tau_2}$. It describes how the abstract input state changes when a piece of code is executed. The set of all abstract denotations from τ_1 to τ_2 is denoted by $\Xi(\tau_1, \tau_2)$. Similarly to the concrete setting, *interpretations* provide abstract denotations for methods in terms of their input and output arguments. An *interpretation* ζ maps method signatures to abstract denotations: $\zeta(\mathsf{mth}) \in \Xi(\mathsf{mth}^i, \mathsf{mth}^i \cup \{out\})$ for every mth. Note that the range of denotations is $\mathsf{mth}^i \cup \{out\}$; this is different from the concrete semantics where only *out* is needed since changes in the memory are directly observable in the heap. The set Ψ includes all abstract interpretations.

4.1. Auxiliary analyses

The abstract semantics uses *purity analysis* [16] as a pre-existent component: a program is supposed to have been analyzed w.r.t. purity using some state-of-the-art tool. The i-th argument of mth is said to be *pure* if mth does not update the data structure to which the argument initially pointed. The analysis proposed by [16], based on [33], can be profitably used: an actual paramater v_i of some mth is said to be possibly impure, written $\dot{v}_i$, if some modification of the data structure accessible from v_i may have been taken place inside mth. Purity information is used when analyzing method calls: if it is possible to prove that v_i is pure, then no new paths in the heap (therefore, no new sharing patterns) related to v_i can have been created in mth.

4.2. Representation of abstract values

This section discusses how elements of the $\bar{\mathcal{I}}_s$ abstract domain can be represented in a logical way: an abstract value I_s can be represented as a unique boolean function $F \in \mathcal{BF}$, as it will be shown in the following. This representation is clearly compatible with the ideas introduced in Section 3, the difference being that all the information about a program point is stored in a single boolean function. The operations employed in the abstract semantics (Sections 4.3, 4.4 and 4.5) will rely on this representation.

The boolean function or propositional formula used to represent an abstract value at a certain program point in a given method mth uses $n + n + m + m$ *propositions*, where n is the number of bits needed to represent all the reference variables in mth, i.e., the ceiling of the logarithm of the number of variables, and

m is the number of reference fields in the program. It is enough to represent all variables in mth, not in the whole program, since variables are local to methods (static fields are not considered here). On the other hand, all reference program fields have to be taken into account in each boolean formula.

Every *model* of such a formula is a truth assignment of all the proposition which describes some possibility of sharing:

- the first n propositions $x_1, .., x_{n-1}$ are the bitwise representation of a program variable v_1;
- the following n propositions $x_n, .., x_{2n-1}$ are the bitwise representation of another program variable v_2;
- the following m propositions $x_{2n}, .., x_{2n+m-1}$, where each x_{2n+k} corresponds to a program field φ_k, represent a path-formula $\langle \omega_1 \rangle$ where $\omega_1 = \{\varphi_k \mid 0 \leq k < m \wedge x_{2n+k} = true\}$; in this sense, each of these propositions corresponds to some f-proposition φ_k.
- the last m propositions represent, similarly, a path-formula $\langle \omega_2 \rangle$ where $\omega_2 = \{\varphi_k \mid 0 \leq k < m \wedge x_{2n+m+k} = true\}$; each of these propositions corresponds to some f-proposition φ_k.

The meaning of such a model is that v_1 and v_2 may F-share where $\langle \omega_1 \cup \omega_2 \rangle$ implies F. In other words, it is possible that there exist converging paths from both variables which traverse exactly the fields corresponding to $\langle \omega_1 \cup \omega_2 \rangle$.

Example 4.1. *Consider a program with six reference fields $\varphi_0, .., \varphi_5$, and a method* mth *with three reference variables v_0, v_1 and v_2; then, the number of propositions needed to represent variables is $n = \lceil \log(3) \rceil = 2$, and the total number of propositions for an abstract value in* mth *is $2+2+6+6 = 16$. Let all the models of a propositional formula be represented as a table, as follows: every column corresponds to a proposition, and every line corresponds to a truth assignments which satisfies the formula. For example:*

1st v.	*2nd v.*	*left*	*right*	
f f	**f f**	**f f f f f f**	**f f f f f f**	*(v_0 and v_0 may share on $\langle \emptyset \rangle$)*
f f	**f f**	**f f t f f f**	**f f t f f f**	*(v_0 and v_0 may share on $\langle \{\varphi_2, \varphi_2\} \rangle$)*
t f	**t f**	**f f f f f f**	**f f f f f f**	*(v_2 and v_2 may share on $\langle \emptyset \rangle$)*

This representation is actually similar to a disjunctive normal form*: every line is a conjunction like $x_0 \wedge \neg x_1 \wedge .. \wedge \neg x_{15}$ (the third line in this table), and all lines are connected by logical disjunction.*

In this representation, the *restriction* $I_s(v_1, v_2)$ which describes the sharing between v_1 and v_2 comes to be $I_s \wedge L_{v_1} \wedge R_{v_2}$, where L_v (resp., R_v) is the formula enconding the bitwise representation of v by using the first n (resp., the second n) propositions. Similarly, $I_s(v, \_) = I_s \wedge L_v$ and $I_s(\_, v) = I_s \wedge R_v$; and $I_s(v)$ is a shorthand for $I_s(v, \_) \vee I_s(\_, v)$.

Example 4.2. *In Example 4.1, let I_s contain the information shown in the table. Then, $L_{v_2} = x_0 \wedge \neg x_1$, and $R_{v_2} = x_2 \wedge \neg x_3$. Therefore, $I_s(v_2, v_2)$ comes to be $\neg x_4 \wedge .. \wedge \neg x_{15}$,*

which corresponds to the truth value of the last $2m$ *propositions in the third line of the table.*

Also, given a path-formula F, the boolean function P_F is defined as the one encoding F by means of the last $2m$ propositions, as mentioned above. For example, $x_{2n} \wedge x_{2n+m} \wedge \neg x_{2n+1} \wedge .. \wedge \neg x_{2n+m-1} \wedge \neg x_{2n+m+1} \wedge .. \wedge x_{2n+2m-1}$ encodes $\left\langle \wp_{\searrow}, \wp_{\swarrow} \right\rangle$, where x_{2n} and x_{2n+m} represent $\wp_{\searrow}$ and $\wp_{\swarrow}$, respectively.

4.2.1. Variable removing and renaming.

A program variable can be *removed* from the sharing information (for example, when it is assigned to **null**) by computing

$$I_s - v \;\; =_{def} \;\; I_s \wedge \neg I_s(v) \;\; = \;\; I_s \wedge \neg(I_s(v, \_) \vee I_s(\_, v))$$

This means that all truth assignments corresponding to v (either as the first or the second variable of the pair) are countermodels of the new formula.

Variable renaming $I_s[v/v']$, which replaces v by v', can be computed as

$$\begin{array}{rcl} (I_s \wedge \neg(I_s(v))) & \vee & \text{[information on other vars, unchanged]} \\ (\exists_{LR}.(I_s(v)))(v', v') & \vee & \text{[from } (v, v) \text{ to } (v', v')] \\ (\exists_L.\, I_s(v, \_))(v', \_) & \vee & \text{[from } (v, w) \text{ to } (v', w)] \\ (\exists_R.\, I_s(\_, v))(\_, v') & & \text{[from } (w, v) \text{ to } (w, v')] \end{array}$$

where $\exists_{LR}$ is the existential cuantification on the first $2n$ propositions; $\exists_L$ cuantifies only on the first n, and $\exists_R$ quantifies on the following n propositions. The first disjunct represents information which is not modified by the renaming (i.e., sharing between other variables); the second disjunct accounts for self-sharing; the third and fourth disjuncts describe how sharing information between v and some w is translated to sharing information between v' and w[4]. To existentially quantify some propositions means making them irrelevant, so that the truth value of $\exists_{LR}.F$ is independent of the truth value of $x_0, .., x_{2n-1}$ (just like the truth of $p \wedge (q \vee \neg q)$ does not depend on the truth value of q).

Example 4.3. *Consider again example 4.1: the abstract value* $I_s - v_2$ *is* $I_s \wedge \neg((x_0 \wedge \neg x_1) \vee (x_2 \wedge \neg x_3))$*, which results in the first two lines of the table. On the other hand, renaming* v_2 *into* v_1 *leads to*

1st v.	*2nd v.*	*left*	*right*	
f f	**f f**	**f f f f f f**	**f f f f f f**	*(*v_0 *and* v_0 *may share on* $\langle\emptyset\rangle$*)*
f f	**f f**	**f f t f f f**	**f f t f f f**	*(*v_0 *and* v_0 *may share on* $\left\langle\{\wp_{\searrow}, \wp_{\swarrow}\}\right\rangle$*)*
f t	**f t**	**f f f f f f**	**f f f f f f**	*(*v_1 *and* v_1 *may share on* $\langle\emptyset\rangle$*)*

[4]Since sharing is a symmetric relation, an implementation would not need to keep both directions (v', w) and (w, v'); however, for clarity, the abstract semantics considers both.

4.2.2. Adding or removing fields

The abstract semantics needs operators to modify boolean functions such that fields are added or removed to their models under certain conditions. In other words, the focus is on modifying formulas from the point of view of *semantics* (w.r.t. their models) instead of syntax (w.r.t. their structure). The following example clarifies this concept and introduces the idea of the $\ominus$ operator, which will be described below.

Example 4.4. *Consider the formula $A = x \wedge (y \vee z)$: its models are $\{x, y\}$, $\{x, z\}$, and $\{x, y, z\}$, where each model is represented by the set of propositions which it interprets as true. The formula $B = \exists z.(A \wedge z)$ has all the models of A where z is true, but z is no longer relevant; in other words, B has models $\{x\}$, $\{x, z\}$, $\{x, y\}$ and $\{x, y, z\}$, and is equivalent to the formula x.*

The *field-addition* operator $\oplus^i : \mathcal{BF} \times \mathcal{F} \mapsto \mathcal{BF}$, for $i \in \{1, 2\}$, works as follows: the formula $F \oplus^1 \varphi$ is satisfied by every model ω of F but also by truth assignments $\omega \cup \{\searrow\!\!\!\!\!\varphi\}$, provided $\swarrow\!\!\!\!\!\varphi$ (the right projection of ω) is the empty set. The new converging paths represented by $F \oplus^1 \varphi$ are those where the right path is empty and the left path is the concatenation of a path represented by F with the field φ. The $\oplus^2$ operator is dual, with the emptyness requirement put on left paths and the concatenation operated on right paths. The meaning of this operator will be clear when discussing the abstract semantics.

In terms of boolean functions, $\oplus_1$ (the dual case $\oplus_2$ is similar) can be computed as

$$F \oplus^1 \varphi = F \vee ((\exists x F) \wedge (\neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}) \wedge x)$$

where: $x \in \{x_{2n}, .., x_{2n+m-1}\}$ is the proposition corresponding to $\searrow\!\!\!\!\!\varphi$; $\exists x F$ makes x irrelevant; the second conjunct is the requirement that $\swarrow\!\!\!\!\!\varphi$ is the empty set; and the conjunction with x includes φ in the left paths.

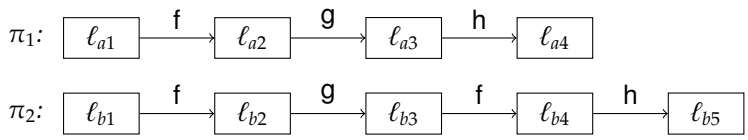
The *path-difference* operator $\ominus : \mathcal{BF} \times \wp(\mathcal{P}) \mapsto \mathcal{BF}$ is defined as follows: given a set X of f-propositions, let Y be the set of corresponding propositions (remember that $\searrow\!\!\!\!\!\varphi$ corresponds to a proposition x_i with $2n \leq i < 2n + m$ while $\swarrow\!\!\!\!\!\varphi$ corresponds to a proposition x_j with $2n + m \leq i < 2n + 2m$). Then

$$F \ominus X = \exists Y \left(F \wedge \bigwedge \{y \mid y \in Y\} \right)$$

The models of the new formula come from those models of F where every proposition in Y is true, but such variables are made irrelevant by the existential quantification as in Example 4.4. Abusing notation, sometimes $F \ominus Y$ will be used directly instead of $F \ominus X$.

The use of this operator is motivated by the following property: let π be $\langle \ell_0, \ell_1.., \ell_k \rangle$ and π' be $\langle \ell_1, .., \ell_k \rangle$; let the sub-path from ℓ_0 to ℓ_1 traverse φ, and the pair (π, π_0) p-satisfy F; then, (π', π_0) p-satisfies $F \ominus \{\searrow\!\!\!\!\!\varphi\}$. Dually, (π_0, π') p-satisfies $F \ominus \{\swarrow\!\!\!\!\!\varphi\}$ whenever (π_0, π) p-satisfies F. This result will be used when dealing with field access.

Example 4.5. *Consider these paths:*



Both pairs (π_1, π_ε)[5] *and* (π_1, π_ε) *p-satisfy* $F = \mathsf{f}_\searrow \wedge \mathsf{g}_\searrow \wedge \mathsf{h}_\searrow$ *since* $\{\mathsf{f}_\searrow, \mathsf{g}_\searrow, \mathsf{h}_\searrow\}$ *is a model of the formula. The definition of path-difference guarantees that both pairs* $(\pi'_1, \pi_\varepsilon)$ *and* $(\pi'_2, \pi_\varepsilon)$*, where* $\pi'_1 = \langle \ell_{a2}, \ell_{a3}, \ell_{a4} \rangle$ *and* $\pi'_2 = \langle \ell_{b2}, \ell_{b3}, \ell_{b4}, \ell_{b5} \rangle$*, p-satisfy* $F \ominus \{\mathsf{f}_\searrow\}$ *since this formula has both* $\{\mathsf{g}_\searrow, \mathsf{h}_\searrow\}$ *and* $\{\mathsf{f}_\searrow, \mathsf{g}_\searrow, \mathsf{h}_\searrow\}$ *among its models. Note that* $\{\mathsf{g}_\searrow, \mathsf{h}_\searrow\}$ *alone is not enough in order to obtain the desired property on field access mentioned above: in fact, this truth assignment is not consistent with* π'_2*, which still traverses the field* f.

Importantly, these operators can be applied not only to path-formulas, but also to generic boolean functions. This is a reason for introducing the representation of abstract information where all the sharing information at a given program point is stored in a unique boolean function: some operations can be applied to all variables with a single use of the logical operators.

Example 4.6. *Suppose the goal is to join*[6] *the abstract sharing information about* v_1 *and any other variable, and the information about* v_2 *and any other variable: this can be obtained by simply computing* $I_s(v_1, \_) \vee I_s(v_2, \_)$ *without having to universally quantify on all program variables.*

4.2.3. "Concatenation" of abstract values

An important operation on propositional formulas models the concatenation of converging paths in the heap: given an abstract value I'_s describing converging paths from (w_1, v), and another abstract value I''_s describing converging paths from (v, w_2), the boolean function $I'_s \odot I''_s$ is meant to model converging paths from (w_1, w_2), provided one of the following two conditions holds: (c1) that the right path of the first pair is empty, or (c2) that the left part of the second pair is. Importantly, the abstract semantics always uses the $\odot$ operator in such a way that either (c1) or (c2) hold.

Example 4.7. *Figure 3 shows how the operator is supposed to work: on the left-hand side,* w_1 *is reaching* v *through a path* π_2*, so that there is a pair of converging paths from* (w_1, w_2) *which is* $(\pi_1 \cdot \pi, \pi_2)$*. On the right-hand side,* v *is reaching* w_2 *and the pair of converging paths from* (w_1, w_2) *is* $(\pi_1, \pi \cdot \pi_2)$*.*

As Example 4.7 points out, the fields traversed by the concatenated path $\pi_1 \cdot \pi$ (respectively, $\pi \cdot \pi_2$) is the union of the sets of fields traversed by π_1 and π (respectively, π and π_2). Therefore, the $\odot$ operator has to guarantee that

[5] Remember that π_ε denotes the empty path, and that the absence of any right f-proposition $\swarrow$ in the path-formula does not impose any condition on the right path.

[6] This operation does not correspond to any rule in the abstract semantics; this example is just for clarity of presentation.

condition (c1) condition (c2)

Figure 3: The meaning of the $\odot$ operator

- The part of sharing information contained in I'_s which is to be "concatenated" refers, as the right variable, to some v which is the same as the left variable referred to by I''_s. In other words, the truth value of the second n propositions in I'_s must be the same as the first n propositions in I''_s.
- As regards the last $2m$ propositions, models of the new formula are the union of models of the operands I'_s and I''_s.

This leads to the following definition:

$$I'_s \odot I''_s = \bigvee_{\omega \in mdls_{\mathcal{P}}(I''_s)} \left(\exists Y_\omega . \left(I'_s \wedge \bigwedge_{i=0}^{n-1} x_{n+i} = \omega(x_i) \right) \wedge \bigwedge_{x \in Y_\omega} x \right)$$

where $Y = \{x_i \mid 2n \leq i \leq 2n{+}2m{-}1 \wedge \omega(x)\}$ is the set of propositions related to fields (i.e., belonging to the last $2m$ propositions) which are true in ω.

The first condition comes from the requirement $\bigwedge_{i=0}^{n-1} x_{n+i} = \omega(x_i)$: each one of the second n propositions in I'_s must have the same truth value as the corresponding proposition in ω, taken from the first n. On the other hand, the second one follows from the way the computation is performed: for every model of I''_s satisfying the first condition, the existential quantification on Y_ω together with $\bigwedge_{x \in Y_\omega} x$ compute the desired set union: if a proposition is true in the considered model ω, then it will be "added" to all relevant models of I'_s.

In the worst case, the computation is expensive since it is necessary to enumerate all models of I''_s; however, in practice, the number of models of an abstract value (which depends on the sharing patterns in the heap) is quite small. In any case, if it is expected that the number of models of I'_s is significantly smaller than the one of I''_s, a symmetric version of this operator can be defined, where the enumeration is done on models of I'_s instead of I''_s.

The $\odot$ operator will be used in dealing with field update in order to model newly-created paths in the heap as the concatenation of existing paths.

4.3. Expressions

Figure 4 describes how the abstract semantics $\mathcal{E}_\zeta[\![\_]\!]$ works on expressions. It is based on a type environment τ (left implicit) and an interpretation ζ on meth-

$$
\begin{array}{lrcl}
(1_e) & \mathcal{E}_\zeta[\![n]\!](I_s) & = & I_s \\
(2_e) & \mathcal{E}_\zeta[\![\textbf{null}]\!](I_s) & = & I_s \\
(3_e) & \mathcal{E}_\zeta[\![\textbf{new}\ \kappa]\!](I_s) & = & I_s \vee (P_{\langle\emptyset\rangle}(\rho,\rho)) \\
(4_e) & \mathcal{E}_\zeta[\![v]\!](I_s) & = & \text{if } \tau(v)\text{=}\textbf{int} \text{ then } I_s \text{ else } I_s \vee I_s[v/\rho] \\
(5_e) & \mathcal{E}_\zeta[\![exp_1 \otimes exp_2]\!](I_s) & = & \mathcal{E}_\zeta[\![exp_2]\!]\left(\mathcal{E}_\zeta[\![exp_1]\!](I_s)\right) \\
(6_e) & \mathcal{E}_\zeta[\![v.\varphi]\!](I_s) & = & \text{if } \varphi \text{ has type } \textbf{int} \text{ then } I_s \text{ else } I_s \vee I_s^a \vee I_s^b \vee I_s^c \text{ where} \\
 & & & I_s^a \;=\; (\exists_{LR}.(I_s(v,v) \ominus \{\searrow, \swarrow\}))(\rho,\rho) \\
 & & & I_s^b \;=\; (\exists_{L}.((I_s(v,\_) \ominus \{\searrow\}) \oplus^2 \varphi))(\rho,\_) \\
 & & & I_s^c \;=\; (\exists_{R}.((I_s(\_,v) \ominus \{\swarrow\}) \oplus^1 \varphi))(\_,\rho)
\end{array}
$$

Figure 4: The abstract semantics for expressions

ods. The special variable ρ represents the result of the evaluation. Each case describes how the boolean formula before a given program point is transferred to the following program point.

4.3.1. Easy cases

Not surprisingly, evaluating an **int** (case 1_e) or a **null** value (case 2_e) does not modify the current abstract value for sharing, i.e., no new sharing paths are created in the heap. In fact, in both cases, ρ has no relation with any existing variable (either because it is a number or because it does not point to a valid location).

On the other hand, when a new object is created (case 3_e), the sharing information does change. ρ is correctly represented by the new abstract value as sharing with itself through a pair of converging empty paths (in fact, only empty paths p-satisfy $\langle\emptyset\rangle$), which means, by the way, that it also aliases with itself. This is the meaning of the formula $P_{\langle\emptyset\rangle}(\rho,\rho)$: its only model is a truth assignment where the first $2n$ propositions correspond to the pair (ρ,ρ), and the rest are all false and represent $\langle\emptyset\rangle$. No existing sharing information is destroyed.

In case 4_e, information about v is copied to ρ, without removing the original information as v is still accessible (this is taken into account by the disjunction between the original I_s and $I_s[v/\rho]$). In case 5_e, $\otimes$ stands for a binary operation on **int**; here, *side effects* are the only possible source of new sharing information. After evaluating each expression, ρ will have an integer value, so that it will never be involved in sharing. Therefore, ρ will never appear in an abstract value.

4.3.2. Field access

Case 6_e is quite harder: if the declared type of φ is a reference type, then the new abstract value is obtained by adding (by disjunction) to the old one the new information $I_s^a \vee I_s^b \vee I_s^c$, as follows. The heap is not modified by the field access, but the frame is: sharing information is assigned to the variable ρ. The formula I_s^a is in charge of representing the newly-computed self-sharing information about ρ; moreover, I_s^b and I_s^c account for the sharing information between ρ and any other variable, including v.

The following lemma contains a detailed explanation of how the rule works, together with a proof of soundness.

Lemma 4.8 (soundness of rule 6_e). *For every state $\sigma = \langle \sigma^f, \sigma^h \rangle$ correctly described by an abstract value I_s (i.e., such that $\sigma \in \bar{\gamma}_s(I_s)$), the state $\sigma' = E^\iota_\tau[\![v.\varphi]\!](\sigma)$ is correctly described by $I'_s = I_s \vee I^a_s \vee I^b_s \vee I^c_s$ computed as in the case 6_e of Figure 4.*

Proof. As a first observation, the heap ${\sigma'}^h$ after accessing φ is identical to σ^h, so that the new abstract value (which is implied by the old one) still describes all relevant paths in the heap. On the other hand, the field access does modify the frame σ^f by assigning to ρ the computed value.

Therefore, soundness amounts to proving that the new abstract value correctly describes the sharing paths involving ρ; concretely, to prove that, under the hypothesis that the previous abstract value I_s correctly describes the heap before accessing the field, (a) I^a_s correctly describes self-sharing about ρ (i.e., sharing about (ρ, ρ)); (b) I^b_s correctly describes sharing about (ρ, w)) for each other variable w, including v; and (c) I^c_s correctly describes sharing about (w, ρ)) for each other variable w, including v. The proof will deal with (a) and (b), while (c) is clearly quite similar to (b) and will not be developed explicitly.

(a) Self-sharing. If $F = I_s(v, v)$ correctly describes the self-sharing information about v, then any pair of paths both starting from $\sigma^f(v)$ and ending in the same location p-satisfies F. The question is: which p-formula will be certainly p-satisfied by every pair of paths starting from ρ? The first observation is that every path from ρ is a sub-path of some path starting from v. Therefore, no path from ρ will traverse fields which were not traversed by a path from v. As a consequence, the new p-formula F' will not need to have models which were not included (by $\subseteq$) in some model of F. Moreover, it can be the case that the field φ is no longer traversed, so that F' needs to contain new models without $\searrow_{\varphi}$ or $\swarrow_{\varphi}$, or both. Finally, paths which provably do not traverse φ will not be super-paths of paths starting from ρ, so that models not including $\searrow_{\varphi}$ and/or (depending on the case, as shown below) $\swarrow_{\varphi}$ are removed. This is exactly what the operator $\ominus$ does in the computation of I^a_s, as explained in Example 4.9 below.

The computation of I^a_s consists of four steps: (1) I_s is restricted to the variable v by computing $I_s(v, v)$; i.e., the new formula only contains information about the self-sharing of v; (2) $\ominus$ is applied in order to account for new paths where φ can be either traversed or not; (3) an existential cuantification is applied on the first $2n$ propositions, in order to make them irrelevant; and (4) the resulting propositional formula is restricted to the pair (ρ, ρ). Steps (3) and (4) together with (1) amount to "transferring" the information from (v, v) to (ρ, ρ), similarly to variable renaming (Section 4.2.1).

It is clear that I^a_s correctly describes the self-sharing about ρ, since $\ominus$ adds to $I_s(v, v)$ those models which represent converging paths from (ρ, ρ), and the operations of steps (3) and (4) assign such information to (ρ, ρ), as expected.

(b) Sharing with other variables. As for sharing between ρ and some other variable w, there are a number of possible scenarios, depending on the previous relation between v and w; the proof will guarantee that I_s^b correctly describes all these possibilities.

The computation of I_s^b consists of five steps: (1) I_s is restricted to the sharing information between v and any other variable by computing $I_s(v, \_)$; (2) since ρ is on a path originating from v and traversing φ, the $\ominus$ operator is applied in order to consider left paths no longer traversing φ, but also paths still traversing φ; (3) the $\oplus^2$ operator accounts for cases where w is reaching v, as explained below; (4) the first n propositions are made irrelevant by existential quantification; and (5) the resulting propositional formula is restricted by requiring that the first variable be ρ. As for I_s^a, steps (4) and (5) together with (1) transfer the sharing information from v to ρ: while $I_s(v, \_)$ contains the sharing between v and any variable, the final proposition contains, after the modifications of steps (2) and (3), the sharing information between ρ and any (other) variable.

The following list takes all scenarios into account: given the existence of a pair of converging paths (π_v, π_w) from (v, w), the proof guarantees that the newly-created pair (π_ρ, π'_w), if any, is correctly described by I_s^b. Given a variable w, there can be different converging paths in the heap (e.g., v could be aliasing with w and also reaching it through a non-empty path). In this case, clearly, I_s^b will account for each one of them because the same operations on propositional formulas are applied in all cases.

- v and w are aliasing, i.e., $\pi_v = \pi_w = \pi_\varepsilon$. In this case, the pair of converging paths corresponds to a path formula with a model {}. No new model is added by step (2) because neither the left nor the right path traverse φ. On the other hand, step (3) adds the model $\{\not\varphi\}$ to the formula which, after steps (4) and (5), will represent the sharing between ρ and w. This is correct because now w reaches ρ by a path $\pi'_w = \left\langle \sigma^f(w), \sigma^f(\rho) \right\rangle$ which traverses φ; i.e., the pair $(\pi_\varepsilon, \left\langle \sigma^f(w), \sigma^f(\rho) \right\rangle)$ is a pair of converging paths from (ρ, w). Note that {} is still a model because $\sigma^f(\rho)$ could be the same location as $\sigma^f(v)$.

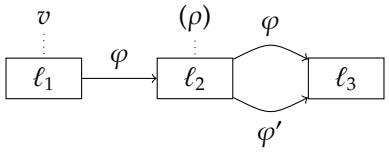
- v reaches w, and ρ is on the path, i.e., π_v is $\left\langle \sigma^f(v), \sigma^f(\rho), .. \sigma^f(w) \right\rangle$ and traverses φ, and $\pi_w = \pi_\varepsilon$. In this case, π_ρ is a subset of π_v which could still traverse φ, and $\pi_w = \pi'_w$. This pair of converging paths is correctly represented by I_s^b because $\ominus\{\not\varphi\}$ considers considers both the traversal and the non-traversal of φ by the left path. Step (3) has no effect in this case because π_v is not empty.

- v reaches w, and ρ is not on the path, i.e., π_v does not contain $\sigma^f(\rho)$. In this case, the pair of converging paths from (v, w) will not generate a pair of converging paths from (ρ, w), so that there is no new information to be taken into account. Note that ρ and w may still share because of other pairs of converging path, but this would be taken into account by other scenarios in this list.

- w reaches v, i.e., $\pi_v = \pi_\varepsilon$. In this case, w will still reach ρ by a path π'_w which is the concatenation of π_w with $\left\langle \sigma^f(v), \sigma^f(\rho) \right\rangle$. In ths case, step (2) adds no models because the left path is not traversing φ. On the other hand, step (3) correctly accounts for the new pair $(\pi_\varepsilon, \pi'_w)$ because, being π_v the empty path, it adds a new model $\{\not\varphi\} \cup \{\not\varphi' \mid \varphi' \text{ is traversed by } \pi_w\}$.
- v and w are sharing on non-empty paths, and ρ is on the left path; i.e., π_v is $\left\langle \sigma^f(v), \sigma^f(\rho), ..\ell \right\rangle$ and traverses φ, and π_w is some non-empty path reaching ℓ. In this case, the new pair (π_ρ, π'_w) of converging paths will be such that $\pi_\rho = \left\langle \sigma^f(\rho), ..\ell \right\rangle$, and $\pi'_w = \pi_w$. This is correctly described by I^b_s since, similarly to the second scenario, $\ominus$ adds a model where the left path no longer traverses φ.
- v and w are sharing on non-empty paths, and ρ is not on the left path. In this case, there is no new pair of converging paths to be considered, and soundness follows easily because there is nothing to prove.
- v and w are the same variable. In this case, all converging paths from (v, v) have to to be studied by considering the previous scenarios (for example, v will be certainly aliasing with itself, so that the first scenario applies).
- v and w are not sharing at all, i.e., there is no such pair of converging paths. In this case, ρ and w will not share either. Soundness follows easily because there is no new pair of converging paths from (ρ, w) to be taken into account.

This completes the proof of soundness. □

Example 4.9. *Consider the following portion of a heap:*



where ρ is the result of evaluating $v.\varphi$. The p-formula satisfied by any pair of converging paths from (v, v) must have, at least, the following models:

$$(1) : \{\},\ \ (2) : \{\not\varphi_{\swarrow}, \not\varphi_{\swarrow}\},\ \ (3) : \{\not\varphi_{\swarrow}, \not\varphi_{\swarrow}, \not\varphi'_{\swarrow}\},\ \ (4) : \{\not\varphi_{\swarrow}, \not\varphi'_{\swarrow}, \not\varphi_{\swarrow}\},\ \ (5) : \{\not\varphi_{\swarrow}, \not\varphi'_{\swarrow}, \not\varphi_{\swarrow}, \not\varphi'_{\swarrow}\}$$

These models correspond, respectively, to: (1) a pair of empty paths; (2) two paths only traversing φ (note that they can have length 1 or 2, depending on whether they reach ℓ_2 or ℓ_3); (3) a pair reaching ℓ_3, with one path traversing the second occurrence of φ, and the other traversing φ'; (4) the dual pair, where φ' is traversed by the first path; and (5) a pair where both paths reach ℓ_3 and traverse φ'.

$$(1_c) \qquad C_\zeta[\![\textbf{skip}]\!](I_s) = I_s$$

$$(2_c) \qquad C_\zeta[\![v{:=}exp]\!](I_s) = (\mathcal{E}_\zeta[\![exp]\!](I_s) - v)[\rho/v]$$

$$(3_c) \qquad C_\zeta[\![v.\varphi{:=}exp]\!](I_s) = (I'_s \vee I^a_s \vee I^b_s \vee I^c_s) - \rho \text{ where}$$
$$I'_s = \mathcal{E}_\zeta[\![exp]\!](I_s)$$
$$I^a_s = (I'_s(\_, v) \wedge \neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}) \odot \diagdown\!\!\!\!\diagdown_{\searrow} \odot I'_s(\rho, \_)$$
$$I^b_s = I'_s(\_, \rho) \odot \swarrow\!\!\!\!\swarrow \odot (I'_s(v, \_) \wedge \neg x_{2n} \wedge .. \wedge \neg x_{2n+m-1})$$
$$I^c_s = (I'_s(\_, v) \wedge \neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}) \odot \diagdown\!\!\!\!\diagdown_{\searrow} \odot I'_s(\rho, \rho) \odot \swarrow\!\!\!\!\swarrow \odot (I'_s(v, \_) \wedge \neg x_{2n} \wedge .. \wedge \neg x_{2n+m-1})$$
$$\diagdown\!\!\!\!\diagdown_{\searrow} = P_{\langle \varphi_{\searrow} \rangle}(v, \rho) \vee \left(\exists_{LR}\left(\exists x_{\varphi_{\searrow}} I'_s(\rho, v) \wedge \varphi_{\searrow} \wedge \neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}\right)\right)(v, \rho)$$
$$\swarrow\!\!\!\!\swarrow = P_{\langle \varphi_{\swarrow} \rangle}(\rho, v) \vee \left(\exists_{LR}\left(\exists x_{\varphi_{\swarrow}} I'_s(v, \rho) \wedge \varphi_{\swarrow} \wedge \neg x_{2n} \wedge .. \wedge \neg x_{2n+m-1}\right)\right)(\rho, v)$$

$$(4_c) \quad C_\zeta\left[\!\!\left[\begin{array}{r}\textbf{if } exp \textbf{ then } com_1 \\ \textbf{else } com_2\end{array}\right]\!\!\right](I_s) = C_\zeta[\![com_1]\!](I_s) \ \vee \ C_\zeta[\![com_2]\!](I_s)$$

$$(5_c) \quad C_\zeta[\![\textbf{while } exp \textbf{ do } com]\!](I_s) = \xi(I_s) \text{ where } \xi \text{ is the least fixpoint of } \lambda w.\lambda I_s.w(C_\zeta[\![com]\!](I_s))$$

$$(6_c) \qquad C_\zeta[\![com_1; com_2]\!](I_s) = C_\zeta[\![com_2]\!](C_\zeta[\![com_1]\!](I_s))$$

$$(7_c) \qquad C_\zeta[\![\textbf{return } exp]\!](I_s) = \mathcal{E}_\zeta[\![exp]\!](I_s)[\rho/out]$$

Figure 5: The abstract semantics for commands

Now, the $\ominus$ operator correctly adds the following models:

$$(6): \ \{\diagdown\!\!\!\!\diagdown_{\searrow}, \varphi'_{\swarrow}\} \quad (7): \ \{\diagdown\!\!\!\!\diagdown'_{\searrow}, \swarrow\!\!\!\!\swarrow\} \quad (8): \ \{\diagdown\!\!\!\!\diagdown'_{\searrow}, \varphi'_{\swarrow}\}$$

since (6) there is a pair of paths from ℓ_2 to ℓ_3 such that the first traverses φ and the second only *traverses φ'; (7) the pair can be reversed*[7]*; and (8) it is also possible that both paths only traverse φ'. However, a number of spurious models are also added:*

$$\{\diagdown\!\!\!\!\diagdown_{\searrow}\}, \ \{\swarrow\!\!\!\!\swarrow\}, \ \{\swarrow\!\!\!\!\swarrow, \varphi'_{\swarrow}\}, \ \{\varphi'_{\swarrow}\}, \ \{\diagdown\!\!\!\!\diagdown_{\searrow}, \diagdown\!\!\!\!\diagdown'_{\searrow}\}, \ \{\diagdown\!\!\!\!\diagdown'_{\searrow}\}, \ \{\diagdown\!\!\!\!\diagdown'_{\searrow}, \swarrow\!\!\!\!\swarrow, \varphi'_{\swarrow}\}, \ \{\diagdown\!\!\!\!\diagdown_{\searrow}, \diagdown\!\!\!\!\diagdown'_{\searrow}, \varphi'_{\swarrow}\}$$

These undesired models have to be considered as a loss of information of the analysis. This example also shows that, in some cases, only one between $\diagdown\!\!\!\!\diagdown_{\searrow}$ and $\swarrow\!\!\!\!\swarrow$ has to be removed (as in the case of $\{\diagdown\!\!\!\!\diagdown_{\searrow}, \varphi'_{\swarrow}\}$, obtained from $\{\diagdown\!\!\!\!\diagdown_{\searrow}, \swarrow\!\!\!\!\swarrow, \varphi'_{\swarrow}\}$).

4.4. Commands

Figure 5 shows how the abstract semantics $C_\zeta[\![\_]\!]$ behaves on commands. The most difficult case, field update, is discussed at the end.

[7] Remember that, as it is presented, the abstract semantics keeps both directions of a pair.

Case 1_c is trivial: nothing changes when **skip** is executed. Case 2_c for variable assignment is also easy: the semantics evaluates *exp*, which could have side effects, and copies the information about ρ to v, after removing the information about v (by "$-v$") since its initial value will be lost. Note that, however, the information about *the location pointed to by* v needs not be (and is not) lost, since there could be other variables pointing to it.

In cases 4_c and 5_c, standard principles for the design of abstract semantics are followed. Both branches of the conditional are analyzed[8]; $C_{\bar{c}}[\![\_]\!]$ is path-insensitive in that the results obtained for each branch are simply combined by means of logical disjunction. In the case of loops, standard fix-point design is used. Termination of the fix-point is guaranteed by the fact that $\bar{\mathcal{I}}_s$ does not allow infinite ascending chains $\langle F_0, .., F_i, .. \rangle$ where $F_j \dot{\sqsubseteq}_s F_{j+1}$ for each $j \geq 0$. However, in principle, there can be chains whose length is exponential on the cardinality of $\mathcal{F}$, so that convergence can be possibly slow unless some mechanism for speeding it up is used (e.g., *widening* [15]).

The last two cases, 6_c and 7_c, are straightforward.

Field update.

The command $v.\varphi$:=*exp* modifies the heap, possibly creating new paths and deleting others. A path π going from $\sigma^f(v)$ to $\sigma^f(\rho)$ and traversing φ is created. The information about π must be *joined* with the original abstract value by means of disjunction.

Given two variables w_1 and w_2, the sharing between them can be modified if π is a sub-path of some newly-created converging paths from (w_1, w_2). This is depicted in Figure 6, and can happen if

(a) there is a path from $\sigma^f(w_1)$ to $\sigma^f(v)$ (i.e., w_1 reaches v), and there is sharing between ρ and w_2;

(b) there is a path from $\sigma^f(w_2)$ to $\sigma^f(v)$, and sharing between w_1 and ρ; or

(c) $\sigma^f(v)$ is reachable from both w_1 and w_2.

Thick lines in the pictures show the new pair of converging paths which is created by the update in each case. Note that sharing information about w_1 (or w_2) and v which is not also reachability information is not considered: only those pairs of converging paths such that the shared location is $\sigma^f(v)$ itself are taken into account; this is consistent with conditions (c1) and (c2) put on the use of the $\odot$ operator in Section 4.2.3. Cases (a), (b) and (c) are captured, respectively, by the propositional formulas I_s^a, I_s^b and I_s^c in rule (3_c) of Figure 5: the new information corresponds to new converging paths shown in Figure 6.

For example, consider case (a), and let the set $\mathbb{X}_s$ be simply $P_{\langle \wp_s \rangle}(v, \rho)$, for the moment. Then, the newly-created pair of converging paths from (w_1, w_2) traverses the following fields: in its left component, fields traversed by π_1,

[8]Recall that, for simplicity, guards are supposed not to have side effects.

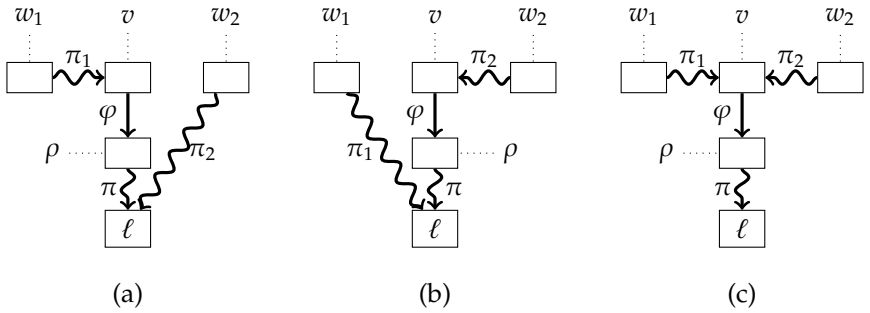


Figure 6: Three possibilities for creating new sharing paths on field update

fields traversed by π, and φ; in its right component, fields traversed by π_2. This is modeled exactly by the formula I_s^a because:

- The first application of $\odot$ satisfies condition (c1) because of the requirement $\neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}$ (it also satisfies (c2), actually).
- The second application of $\odot$ also satisfies (c1) because of $\boxtimes$ (this will still hold when the second component of $\boxtimes$ will be considered).
- The first operand describes converging paths between any variable w_1 and v, provided the right path is empty; the second operand describes converging paths from v to ρ, with the same condition on right paths; and the third operand describes converging paths from ρ to some variable w_2; therefore, due to the definition of $\odot$, the final I_s^a describes converging paths from any pair (w_1, w_2), as desired.

A similar discussion holds for cases (b) and (c).

There is a last possibility to be considered: that the update of φ closes a cycle between ρ and v. In fact, suppose that there was a path from $\sigma^f(\rho)$ to $\sigma^f(v)$: in such a case, the field update will create a cycle which can be possibly traversed by converging paths from (w_1, w_2). This contingency is captured by the second part of the definition of $\boxtimes$ and $\not\boxtimes$: instead of only traversing φ, newly created paths may also traverse the whole cycle. Figure 7 shows this situation in case (a). Here, the newly-created pair of converging paths does the following: it goes (1) from $\sigma^f(w_1)$ to $\sigma^f(v)$; (2) from $\sigma^f(v)$ to $\sigma^f(\rho)$; (3) backwards from $\sigma^f(\rho)$ to $\sigma^f(v)$; (4) again, from $\sigma^f(v)$ to $\sigma^f(\rho)$; and (5) from $\sigma^f(\rho)$ to ℓ. Therefore, there are actually eight cases to be considered: two related to (a), two related to (b), and four related to (c) (the latter holds because both $\boxtimes$ and $\not\boxtimes$ come into play).

Example 4.10 (field update). *Consider the execution of the assignment* `x.f := y.h` *in an initial state which is correctly approximated by the following abstract value* I_s

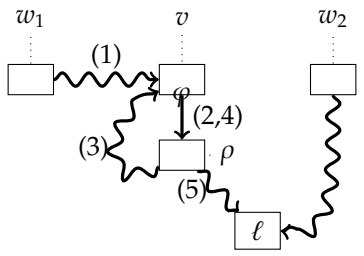
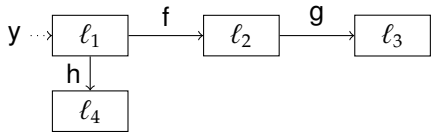


Figure 7: Field updates which close a cycle in case (a)

(only relevant information is displayed):

$$
\begin{array}{lcll}
I_s(\mathsf{x},\mathsf{x}) & = & \langle\emptyset\rangle & \textit{(}\mathsf{x}\textit{ is not null but all its fields are)} \\
I_s(\mathsf{z},\mathsf{x}) & = & \langle\emptyset\rangle & \textit{(aliasing is the only possible relation between }\mathsf{x}\textit{ and }\mathsf{z}\textit{)} \\
I_s(\mathsf{x},\mathsf{y}) & = & \left\langle \swarrow\!\mathsf{f}, \swarrow\!\mathsf{g} \right\rangle & \textit{(the only possible path from }\mathsf{y}\textit{ to }\mathsf{x}\textit{ traverses }\mathsf{f}\textit{ and }\mathsf{g}\textit{)} \\
I_s(\mathsf{y},\mathsf{x}) & = & \left\langle \searrow\!\mathsf{f}, \searrow\!\mathsf{g} \right\rangle & \textit{(sharing is symmetric)} \\
I_s(\mathsf{y},\mathsf{y}) & = & \left\langle \searrow\!\mathsf{f}, \swarrow\!\mathsf{f} \right\rangle \vee \left\langle \searrow\!\mathsf{f}, \searrow\!\mathsf{g}, \swarrow\!\mathsf{f}, \swarrow\!\mathsf{g} \right\rangle \vee \left\langle \searrow\!\mathsf{h}, \swarrow\!\mathsf{h} \right\rangle \vee \langle\emptyset\rangle &
\end{array}
$$

The last line may represent a heap portion like, for example, the following:



The abstract semantics first evaluates the expression $\mathsf{y.h}$*, giving a new abstract value* I'_s *such that:*

$$
\begin{array}{lcll}
I'_s(\rho,\rho) & = & \left\langle \searrow\!\mathsf{h}, \swarrow\!\mathsf{h} \right\rangle \vee \left\langle \searrow\!\mathsf{h} \right\rangle \vee \left\langle \swarrow\!\mathsf{h} \right\rangle \vee \langle\emptyset\rangle & \textit{(not involving }\mathsf{f}\textit{ or }\mathsf{g}\textit{)} \\
I'_s(\mathsf{y},\rho) & = & \left\langle \searrow\!\mathsf{h}, \swarrow\!\mathsf{h} \right\rangle \vee \left\langle \searrow\!\mathsf{h} \right\rangle & \\
I'_s(\rho,\mathsf{x}) & = & \textit{false} & \textit{(nothing is added)}
\end{array}
$$

where (a) the second line correctly allows some non-trivial sharing (not just reachability through h*) between* y *and* ρ *because the initial information describing* y *is compatible with paths in the heap which are not displayed in the picture above (e.g., a cycle from* ℓ_4 *to itself, only traversing* h*); and (b) the last line holds because* ρ *is certainly out of any path from* $\sigma^f(\mathsf{y})$ *to* $\sigma^f(\mathsf{x})$*. Finally, the final abstract state* I''_s *is computed by applying the abstract semantics to the field update. The most relevant information (assuming that* f*,* g *and* h *are the only fields) is:*

$$
\begin{array}{lcl}
I''_s(\mathsf{x},\mathsf{x}) & = & \searrow\!\mathsf{f} \wedge \swarrow\!\mathsf{f} \wedge \neg\searrow\!\mathsf{g} \wedge \neg\swarrow\!\mathsf{g} \\
I''_s(\mathsf{x},\mathsf{y}) & = & \searrow\!\mathsf{f} \wedge \neg\searrow\!\mathsf{g} \wedge (\swarrow\!\mathsf{f} \leftrightarrow \swarrow\!\mathsf{g}) \wedge (\neg\swarrow\!\mathsf{f} \rightarrow \swarrow\!\mathsf{h})
\end{array}
$$

The models of the first path-formula are

$$
\left\{\searrow\!\mathsf{f}, \searrow\!\mathsf{h}, \swarrow\!\mathsf{h}, \swarrow\!\mathsf{f}\right\} \quad \left\{\searrow\!\mathsf{f}, \searrow\!\mathsf{h}, \swarrow\!\mathsf{f}\right\} \quad \left\{\searrow\!\mathsf{f}, \swarrow\!\mathsf{h}, \swarrow\!\mathsf{f}\right\} \quad \left\{\searrow\!\mathsf{f}, \swarrow\!\mathsf{f}\right\}
$$

while the second has these models (the first two coming from case (a) of rule (3_c)*, the others coming from case (c)):*

$$\{\nearrow f, \searrow h, \swarrow h\} \; \{\nearrow f, \swarrow h\} \; \{\nearrow f, \searrow h, \swarrow f, \swarrow g, \swarrow h\} \; \{\nearrow f, \searrow h, \swarrow f, \swarrow g\} \; \{\nearrow f, \swarrow f, \swarrow g, \swarrow h\} \; \{\nearrow f, \swarrow f, \swarrow g\}$$

Note that, in this example, $\nwarrow\!\!\!\!\searrow$ *and* $\swarrow\!\!\!\!\nearrow$ *only consider* $\nearrow f \searrow$ *and* $\swarrow f \nearrow$*, respectively, since there is no path from* ρ *to* x*. The final result is that, for instance, every pair of converging paths from* (x, y) *must traverse* f *but not* g *in the left side, and may alternatively traverse both* f *and* g *or* h *in the right side.*

Lemma 4.11 (soundness of rule 3_c**).** *For every state* $\sigma = \langle \sigma^f, \sigma^h \rangle$ *correctly described by* I_s *(i.e., such that* $\sigma \in \bar{\gamma}_s(I_s)$*), the state* $\sigma' = C^{\iota}_{\tau}[\![v.\varphi{:=}exp]\!](\sigma)$ *is correctly described by* $I''_s = (I'_s \vee I^a_s \vee I^b_s \vee I^c_s) - \rho$ *computed as in the case* 3_c *of Figure 5.*

Proof. Most of the argument for soundness has been already developed in the description of rule 3_c. In particular:

- That the application of the $\odot$ operator accounts for the concatenation of converging paths as described in Section 4.2.3.
- That, consequently, formulas I^a_s, I^b_s and I^c_s describe converging paths from any two variables w_1 and w_2 in the three scenarios described in Figure 6.

The second result comes from the way abstract values are represented as propositional formulas, as already mentioned before: in the definition of I^a_s, the boolean function $I'_s(\_, v)$ describes the sharing of v with any variable, including itself, as a single formula; similarly, $I'_s(\rho, \_)$ describes the sharing of ρ with any variable, including itself.

It remains to prove that the three scenarios of Figure 6 cover all cases. Note that sharing about (w_1, v) is covered by case (a) when π_1 is empty; sharing about (v, w_2) is covered by case (b) when π_2 is empty; and self-sharing about v is covered by case (c) when both π_1 and π_2 are empty. For the same reason, aliasing between v and any variable is also covered.

Moreover, if both w_1 and w_2 only share with v, but do not reach it nor are reached by it, then they do not share with each other after the update unless they were already sharing before. In other words, there is no way to create a pair of converging paths from (w_1, w_2) if v is not included in the left or the right path, which amounts to say that either w_1 or w_2 (or both) are reaching v or aliasing with it.

Therefore, rule 3_c correctly describes all newly-created paths in the heap; moreover, it keeps all previous information, including the one about paths which have been broken by the update. This is a matter of precision, not soundness; in general, the information contained in abstract values is not enough to recognize when a path is guaranteed to be broken by a field update. □

$$
\begin{array}{l}
(7_e)\ \mathcal{E}_\zeta[\![v_0.mth(v_1,..,v_n)]\!](I_s) = \quad I_s \ \vee\ I_s'''' \text{ where } \bar{v}=\{v_0,..,v_n\} \text{ and} \\
\qquad I_s' = I_s - (\tau\backslash\bar{v}) \\
\qquad I_s'' = \vee\ \{\ (\zeta(\mathsf{mth})(I_s'[\bar{v}/\mathsf{mth}^i]))[\mathsf{mth}^i/\bar{v}, out/\rho]\ \mid\ \mathsf{mth} \text{ can be called here }\ \} \\
\qquad I_s^{ij} = (\dot{v}_i \vee \dot{v}_j) \wedge \left(\begin{array}{l} ((\exists x_{2n+m}..x_{2n+2m-1} I_s(\_, v_i)) \wedge \neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}) \\ \odot\ \exists_{LR}\ F_{ij}^m \\ \odot\ ((\exists x_{2n}..x_{2n+m-1} I_s(v_j, \_)) \wedge \neg x_{2n} \wedge .. \wedge \neg x_{2n+m-1}) \end{array} \right) \\
\qquad\qquad F_{ij}^0 \ = \ G_{ij}^m \\
\qquad\qquad F_{ij}^{k+1} \ = \ \textsc{ite}\left(I_s(v_j,\_) \wedge x_{2n+k},\ F_{ij}^k \ominus \{x_{2n+m+k}\},\ F_{ij}^k\right) \\
\qquad\qquad G_{ij}^0 \ = \ I_s''(v_i, v_j) \\
\qquad\qquad G_{ij}^{k+1} \ = \ \textsc{ite}\left(I_s(\_, v_i) \wedge x_{2n+m+k},\ G_{ij}^k \ominus \{x_{2n+k}\},\ G_{ij}^k\right) \\
\qquad I_s''' = \vee\ \{\ I_s^{ij} \mid i, j \in \{0..n\}\} \\
\\
\qquad I_s^i = ((\exists x_{2n+m}..x_{2n+2m-1} I_s''(\rho, v_i)) \wedge \neg x_{2n+m} \wedge .. \wedge \neg x_{2n+2m-1}) \odot H_i^m \\
\qquad\qquad H_i^0 \ = \ I_s(v_i, \_) \\
\qquad\qquad H_i^{k+1} \ = \ \textsc{ite}\left(I_s''(\rho, v_i) \wedge x_{2n+m+k},\ H_i^k \ominus \{x_{2n+k}\},\ H_i^k\right) \\
\qquad I_s'''' = I_s''' \vee \left(\vee_{0\le i\le n}(I_s^i)\right)(\rho, \_) \qquad\qquad \text{(the case } (\_, \rho) \text{ is dual)}
\end{array}
$$

Figure 8: The abstract semantics for method calls

4.5. Method calls

Case 7_e in Figure 8 describes the behavior of the abstract semantics on method calls, which are considered as expressions. For simplicity, methods without return value are not included in the language; however, they could be easily dealt with by slightly modifying the semantics. As usual in Object-Oriented programs, a reference variable v with declared type $\delta(v) = \kappa$ may store at runtime any object of type $\kappa' \le \kappa$, where $\le$ is the subclass relation. The set of possible runtime types of v can be computed statically by *class analysis* [34] whenever needed; if such an analysis is not available, then this set can be taken, conservatively, to be $\{\kappa \mid \kappa \le \delta(v)\}$.

First, the abstract value I_s' is obtained by restricting the initial abstract value to the actual parameters $\bar{v}$ of mth. Also, I_s'' comes from applying the denotation of mth for sharing.

Afterwards, for every two actual parameters v_i and v_j, a new formula I_s^{ij} is computed. If both v_i and v_j are pure in the invoked method, then $I_s^{ij} = \mathit{false}$ as required by "$(\dot{v}_i \vee \dot{v}_j) \wedge ...$"; in fact, purity implies that no modification to their corresponding data structures took place during the method execution, so that no new sharing paths can be created by modifying v_i or v_j, which involve any two variables w_1 and w_2 outside mth. As discussed in Section 4.1, purity analysis is taken as pre-computed information.

Otherwise, the possible impurity of v_i or v_j implies that a new pair of converging paths from (w_1, w_2) could have been created inside mth. Such paths will have the following structure: the left path π_1 goes from w_1 to a location ℓ_1 which is reachable from both w_1 and v_i; from there, it goes to some location

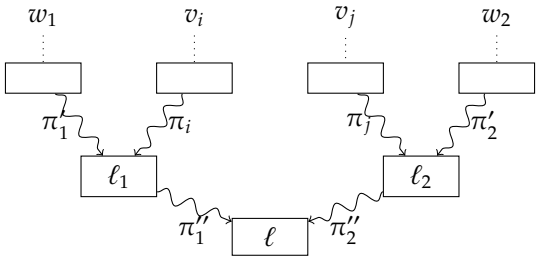


Figure 9: Creation of new sharing patterns during a method invocation

ℓ. On the other hand, the right path π_2 goes from w_2 to a location ℓ_2 which is reachable from both v_j and w_2; from there, it converges to the same location ℓ. Therefore, (1) the first part of π_1 is described by the sharing information $I_s(w_1, v_i)$ in its left component; (2) the first part of π_2 is described by $I_s(v_j, w_2)$ in its right component; and (3) the second part of both paths is (not trivially) represented by $I''_s(v_i, v_j)$.

Consider Figure 9: here, w_1 and w_2 are variables visible in the invoking method, and v_i and v_j are actual parameters, also visible in the invoking method. The execution of mth may create a new sharing pattern between w_1 and w_2 if either v_i or v_j are impure, i.e., if there can be some modification in the data structures reachable from the actual parameters. Such a sharing pattern would be a pair of converging paths from (w_1, w_2) such that the left component $\pi_1 = \pi'_1 \cdot \pi''_1$ goes through location ℓ_1, the right component $\pi_2 = \pi'_2 \cdot \pi''_2$ goes through ℓ_2, and both reach ℓ.

The propositional formula I_s^{ij} is in charge of describing the new sharing patterns corresponding to v_i and v_j, for every w_1 and w_2. The formula is made of three parts, related by the $\odot$ operator.

- The first part describes π'_1, i.e., converging paths from w_1 and the location ℓ_1. Since, in principle, there is no variable pointing directly to ℓ_1, it is necessary to remove the right component from $I_s(\_, v_i)$, corresponding to π_i: this is obtained by the existential quantification and the negation of all variables corresponding to the right path. By doing this, the sharing information no longer describes converging paths from (w_1, v_i); instead, converging paths starting from $\sigma^f(w_1)$ and ℓ_1 are represented, and the right component is the empty path.

- The second part describes converging paths from ℓ_1 and ℓ_2. This information is obtained from $I''_s(v_i, v_j)$, which describes paths $\pi_i \cdot \pi''_1$ and $\pi_j \cdot \pi''_2$. The goal here is to describe correctly π''_1 and π''_2: the fields traversed by π''_1 (the dual case is similar) are those traversed by $\pi_i \cdot \pi''_1$, except, possibly, those traversed by π_i. To say that traversing a field is no longer

necessary, but still possible, one has to apply the $\ominus$ operator[9]. This leads to the construction of F^m_{ij}, a propositional formula defined recursively which possibly removes from paths all the propositions which are true in the left component of $I_s(w_1, v_i)$, i.e., corresponding to fields which can be traversed by π_i. As usual, the *conditional* formula $\text{ITE}(A,\ B,\ C)$ is B or C depending on the value of A, i.e., $\text{ITE}(A,\ B,\ C) \equiv (A \wedge B) \vee (\neg A \wedge C)$. Then, the definition of F^{k+1}_{ij} works on x_{2n+m+k} of the recursive call, corresponding to some $\swarrow\!\!\!\!\!\varphi$, if x_{2n+k}, corresponding to $\searrow\!\!\!\!\!\rho$, is true in $I_s(v_j, \_)$. In other words, $\ominus$ is applied to a proposition in the *right* component of $I''_s(v_i, v_j)$ if it appears in the *left* component of $I_s(v_j, \_)$, i.e., in π'_2. The same happens in G^m_{ij} which accounts for the left component of $I''_s(v_i, v_j)$. Importantly, the resulting formula $\exists_{LR} F^m_{ij}$ does not refer to any specific variables (it is, so to say, a pure path-formula only describing fields). This way, it can be combined by $\odot$ with any formula.

- Similarly to the first part, the third part describes π'_2, i.e., converging paths from ℓ_2 and w_2.

To connect all parts of the formula by $\odot$ gives the expected information: the result of concatenating

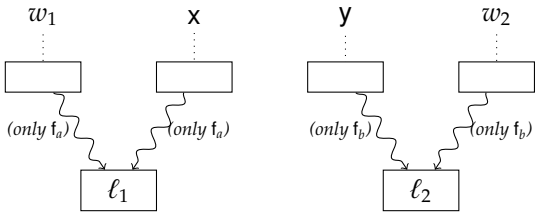
- converging paths from w_1 and ℓ_1;
- converging paths from ℓ_1 and ℓ_2; and
- converging paths from ℓ_2 and w_1

describes converging paths from w_1 and w_2. Note that condition (c1) of the $\odot$ operator is satified in the first application, and condition (c2) is satified in the second application.

It is clear that, as a propositional formula, I^{ij}_s can be syntactically quite large and complex. However, representations of boolean formulas like Binary Decision Diagrams would allow to perform these operations efficiently.

Example 4.12. *Let* $\mathsf{x}.mth(\mathsf{y})$ *be a method call with two parameters* x *(the called object) and* y *(the unique argument). Let* w_1 *and* w_2 *be two variables visible outside mth, and* $I_s(w_1, \mathsf{x})$ *be* $\langle \swarrow\!\!\!\!\!\mathsf{f}_b, \swarrow\!\!\!\!\!\mathsf{f}_a \rangle$*, meaning that any pair of converging paths from* (w_1, x) *only traverses* f_a*. Similarly,* $I_s(\mathsf{y}, w_2) = \langle \swarrow\!\!\!\!\!\mathsf{f}_b, \swarrow\!\!\!\!\!\mathsf{f}_b \rangle$*, as shown in the picture below. Suppose also that* $I''_s(\mathsf{x}, \mathsf{y}) = \left\langle \searrow\!\!\!\!\!\mathsf{g}, \swarrow\!\!\!\!\!\mathsf{g} \right\rangle$ *describes the sharing between* x *and* y *created inside mth.*

[9]Slightly abusing notation, a set of propositions is used as the second argument of $\ominus$ instead of a set of f-propositions.



Clearly, no new sharing between w_1 and w_2 can be created in this case since:

- *according to $I_s''(x, y)$, new converging paths from (x, y) to some location ℓ can only traverse g; and*
- *in order to affect the sharing between w_1 and w_2, such converging paths should go through ℓ_1 (the left one) and ℓ_2 (the right one); but*
- *a path going from x to ℓ through ℓ_1 would be forced to traverse f_a according to the initial sharing information; and*
- *similarly, a path from y to ℓ through ℓ_2 would be forced to traverse f_b;*
- *this is not possible since the sharing information $I_s''(x, y)$ does not allow the traversal of f_a or f_b.*

Therefore, if there are some new converging paths from (x, y), then they cannot include any location shared with w_1 or w_2, which is equivalent to say that the sharing between w_1 and w_2 is not affected by mth. This situation is captured by the abstract semantics since $\ominus$ requires that, in order to be existentially quantified, a proposition must be true in the formula. In this case, the propositions corresponding to f_b are false in $I_s''(x, y)$, so that the final value of F_{01}^m is false.

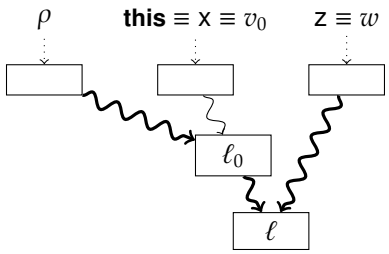
All the newly-computed I_s^{ij} are combined into I_s''' by disjunction. Unlike previous work [38], this sharing analysis is able to deal precisely with *cutpoints* [29], i.e., an objects which are (a) reachable from a parameter of *mth* in at least one step; and (b) also reachable by traversing a path which does not include any object which is reachable from any parameter of *mth*. For example, objects at ℓ_1 and ℓ_2 in Figure 9 could be cutpoints (condition (b) could be true or false depending on other actual parameters). The more precise behavior of the present approach comes from the fact that the previous reachability and cyclicity analysis was using standard, field-insensitive sharing to deal with such cases, thus losing some field-sensitive information while processing the method call.

The next step is to propagate the information about some v_i in I_s''' to ρ whenever v_i shares with ρ after the call. This is needed in order to take into account some cases similar to Example 4.13.

Example 4.13 (return value). *Consider the following fragment [38], where κ is some declared class:*

```
1 κ m() {
2   κ a; a := new κ;
3   a.f := this.f;
4   return a; }
5 y.g := z;
6 x.f := y;
7 w := x.m();
```

After line 6, the computed abstract value I_s^6 *is such that* $I_s^6(\mathsf{x},\mathsf{z}) = \langle \mathsf{f}_\searrow, \mathsf{g}_\searrow \rangle$. *After line 7,* w *shares with* z *because the sharing between* w *and* x *is created in* m, *and* x *was reaching* z. *However, this sharing information cannot be produced by applying the denotation of* m, *since* z *is not a parameter (so that* m *does not know of its existence). In fact, the denotation of* m *can only detect that* ρ *and* **this** *share at the end of* m, *i.e., that* w *and* x *share after the call. The only way for the abstract semantics to be sound is to propagate the sharing information between* x *and* z *to the sharing between* ρ *and* z, *which is then copied into the sharing between* w *and* z. *Note that* **this** *(i.e., the actual parameter* x *which corresponds to the formal parameter* v_0 *in the signature) is pure in* m, *so that no new model is added to* I_s^{00} *(which is the only* I_s^{ij} *to be considered here). The propagation is obtained by building the converging paths from* (ρ, z) *and from* (z, ρ) *as a modification of converging paths from* (x, z). *The following picture gives an idea of the rationale behind such a propagation:*



Each pair of converging paths from (ρ, z) *which are created from the sharing between* ρ *and* v_0 *are computed by taking converging paths from* (v_0, z), *and removing from the left path the fields (only a subset X of them, similarly to Example 4.5) involved in the path from* v_0 *to* ℓ_0, *and adding fields involved in the path from* ρ *to* ℓ_0.

The definition of each H_i follows this intuition: in order to describe the sharing between ρ and some w (in the example, z), it is necessary to

- From converging paths from (ρ, v_i), remove information about the right path (this is done in I_s^i by existentially quantifying and restricting on variables $x_{2n+m}, \ldots, x_{2n+2m-1}$, as in I_s^{ij}); this gives information about paths from $\sigma^f(\rho)$ to the shared location ℓ_i (ℓ_0 in the picture).
- From converging paths from (v_i, w), possibly remove from the left component those fields which are on the path from $\sigma^f(v_i)$ to ℓ_i, in order to model paths from ℓ_i to ℓ; this is obtained by applying $\ominus$ in the computation of H_i^m, whose structure is similar to F_{ij}^m and G_{ij}^m.
- Joining both parts by $\odot$ (note that condition (c1) holds), in order to describe the desired sharing patterns.

Lemma 4.14 (soundness of rule 7_e). *For every state $\sigma = \left\langle \sigma^f, \sigma^h \right\rangle$ correctly described by I_s (i.e., such that $\sigma \in \bar{\gamma}_s(I_s)$), the state $\sigma' = E^{\iota}_{\tau}[\![v_0.mth(v_1, \ldots, v_n)]\!](\sigma)$ is correctly described by $I_s \vee I_s''''$ computed as in the case 7_e of Figure 8.*

Proof. The resulting abstract state contains in I_s all the sharing information before the call; therefore, the proof consists of demonstrating that I_s'''' correctly describes all the newly-created sharing patterns. First, the creation of new converging paths by modifying actual parameters is taken into account; afterwards, the return value is also considered.

Actual parameters. The first observation is that converging paths from any two variables w_1 and w_2 of the invoking method (which could possibly be actual parameters) can be created during the execution of mth only if there exist two actual parameters v_i and v_j such that:

(a) mth modifies the data structure accessible from one of them, or both data structures; otherwise, the heap which is visible from the invoking method is not changed after the call (so that there are no new sharing patterns to be described) since any modification of the heap which is local to mth cannot be seen from outside (note that information can only be passed through actual parameters since there are no global variables). Therefore, if it is possible to prove that both v_i and v_j are *pure* (Section 4.1), then there is no need to search for new sharing patterns.

(b) w_1 shares with v_i and w_2 shares with v_j. In fact, if w_1 and w_2 do not share with any actual parameters, then no modification of the heap in mth will be relevant to them, and no sharing between them will be created.

(c) Modifications of the heap occur in mth, such that

- either the data structure accessible from ℓ_1 or the data structure accessible from ℓ_2 (or both) are modified, where ℓ_1 is a location shared by w_1 and v_i before the call, and ℓ_2 is a location shared by v_j and w_2 before the call; note that, consistently with condition (a), this implies the impurity of at least one between v_i and v_j; and
- such modifications make ℓ_1 and ℓ_2 sharing with each other by the creation of a new pair of converging paths reaching some ℓ.

Condition (a) is used by stating that $I_s^{ij} \neq \mathit{false}$ only if either v_i or v_j are impure: this comes from the condition "$(\dot{v}_i \vee \dot{v}_j) \wedge \ldots$" in the definition of I_s^{ij}. Moreover, the abstract information $I_s(\_, v_i)$ represents all sharing involving v_i: if w_1 does not share with v_i, then $I_s(w_1, v_i) = \mathit{false}$, which means that no models of $I_s(\_, v_i)$ will refer to w_1 as the first variable. The same happens with v_j and w_2; in the end, as condition (b) guarantees, I_s^{ij} will have models involving w_1 and w_2 only if they share with some actual parameters.

As for condition (c), the propositional formula I_s^{ij} is in charge of describing such new converging paths from (w_1, w_2) by using the previous information

about the sharing between w_1 and v_i and between v_j and w_2, together with a description $I_s''(v_i, v_j)$ of the new sharing patterns between v_i and v_j created inside mth. Concretely, consider Figure 9: since $I_s''(v_i, v_j)$ describes converging paths like $\pi_i \cdot \pi_1''$ and $\pi_j \cdot \pi_2''$, $I_s(\_, v_i)$ describes converging paths like π_1 and π_i for any w_1, and $I_s(v_j, \_)$ describes converging paths like π_j and π_2 for any w_2, then the goal of F_{ij}^m is to describe converging paths like π_1'' and π_2''. To this end, F_{ij}^m applies the $\ominus$ operator as described before, which results in describing sub-paths $\pi_i = (\pi_i \cdot \pi_1'') - \pi_1''$ and $\pi_j = (\pi_j \cdot \pi_2'') - \pi_2''$ where the "minus" symbol is path-difference.

Importantly, the computed propositional formulas account for every w_1 and w_2, including actual parameters, without mentioning them explicitly. Moreover, each possible path in the heap is included: given specific values for the sharing information $I_s(v_j, \_)$ and $I_s(\_, v_i)$, the computation of F_{ij}^m builds a formula that applies $\ominus$ on each proposition depending on its value in each model of $I_s(v_j, \_)$ and $I_s(\_, v_i)$.

Return value. A similar discussion holds for the return value of mth. Consider the figure in Example 4.13: each abstract value I_s^i describes the concatenation of two pairs of converging paths: the ones from ρ and ℓ_0 (one of them is empty as required by $\odot$), and the ones from ℓ_0 and w (z in the picture):

- paths from ρ and ℓ_0 can be easily modeled by taking $I_s''(\rho, v_i)$ and removing the right component by existential quantification and restriction, as in the definition of I_s^i; and
- paths from ℓ_0 and w can be modeled, as before, by considering the path from ℓ_0 to ℓ as the path-difference between the path from v_i to ℓ and the one from v_i to ℓ_0; this is exactly what H_i^m does.

The rest of the sharing involving ρ (for example, sharing with actual parameters) is described directly by I_s''. □

4.6. Global fix-point

Not surprisingly, the abstract semantics of a program P is the *least fix-point* of an (abstract) transformer of interpretations. The *abstract denotational semantics* of a program P is quite standard: it is the *lfp* of

$$\mathcal{T}_P(\zeta) = \left\{ \mathsf{mth} \mapsto \lambda I_s \in \bar{\mathcal{I}}_s^{\mathsf{mth}^i} \left(\exists X. C_\zeta \llbracket \mathsf{mth}^b \rrbracket (I_s \vee I_s[\bar{w}/\bar{u}]) \right) [\bar{u}/\bar{w}] \mid \mathsf{mth} \in P \right\}$$

where $\mathsf{mth}^i = \{this, w_1, \ldots, w_n\}$, and $\bar{u}$ is a variable set $\{u_1, \ldots, u_n\}$ such that $\bar{u} \cap \mathsf{mth}^s = \emptyset$; moreover, $\mathrm{dom}(\tau) = \mathsf{mth}^l \cup \bar{u}$, and $X = \mathrm{dom}(\tau) \backslash (\bar{u} \cup \{this, out\})$. Variables $\bar{u}$ play the role of *ghost variables*: they are used to keep a copy of the original information about formal parameters, in order to keep track of data structures to which input variables point at the beginning of a method, since otherwise they could be lost if the corresponding variables are updated. For example, if a formal parameter is updated by some v:=*exp* command, it does not

mean that the corresponding data structure is lost: in fact, it may be still accessed from outside the invoking method, for example from the corresponding actual parameter.

5. Practical issues

This field-sensitive sharing analysis has been implemented[10] as a tool based on the Chord Java bytecode analyzer [25]. The implementation covers most sequential Java bytecode instructions which may occur in the single method the analysis focuses on. A path-formula is explicitly represented as the set of its models, which are, in turn, sets of f-propositions. During the fix-point computation, path-formulæ are always combined by disjunction, which amounts to add new sets of f-propositions (i.e., models) to their representation. This internal representation of sharing information is different from the one described in Section 4, and more similar to previous work [38]. Ongoing and future work includes shifting to the representation of Section 4 by using efficient tools like the JavaBDD library [36] for Binary Decision Diagrams; actually, the new representation of sharing information has been introduced with BDDs in mind. The current implementation is already partially prepared for this change, and part of it has already been done: exploiting the features of Object-Oriented programming, the source code is organized in such a way that a different internal representation can be added with a reasonable effort and without having to rewrite all the code. The rest of this section discussed issues related to the current implementation (with sets of models); hopefully, most aspects will be improved when using BDDs.

Needless to say, the analyzer takes Java bytecode instead of Java source code. Relying on Chord, the implementation is *context-sensitive*; in particular, it is k-object-sensitive with $k = 1$. When dealing with method calls, *ghost variables* are used in order to store the initial variable of parameters in the invoked method; this allows to keep important information in case a formal parameter is strongly updated (i.e., it no longer points to the data structure passed as actual parameter) during the execution.

The number of truth assignments of a path-formula is exponential on the number of fields in the program: there are $2^{2|\mathcal{F}|}$ possible truth assignments in a program with $|\mathcal{F}|$ declared fields. However, a truth assignment can be easily represented as a *bit vector*, so that a path-formula is a set of bit vectors: its models. This way, operations on path-formulæ can be carried out quite efficiently by using existing technology.

To represent a path-formula F by a set of bit vectors becomes impracticable if F has a large number of models. However, even if $|\mathcal{F}|$ is a fairly large number (which implies that bit vectors are long), it is highly unlikely that a path-formula has a big number of models: in fact, it is usually the case that

[10] Available at `http://costa.ls.fi.upm.es/~damiano/fSharing/` and in GitHub at `https://github.com/damianozanardini/java-heap-analyzer`.

converging paths at some point of the execution only involve a small subset of $\mathcal{F}$, so that the number of models is reduced with respect to the upper bound $2^{2|\mathcal{F}|}$, and bit vectors are likely to contain much more "0s" than "1s". For example, if the creation of a double-linked list is analyzed, then converging paths will probably only involve the *next* and *prev* fields, regardless of whether the list is created as a part of a much bigger program.

Anyway, the efficiency of the static analysis presented here can be significantly improved if one knows for what the sharing analysis will be used (remember that sharing information is often used for other analyses like termination or shape analysis). In fact, suppose that a portion of code is analyzed because it is critical to some desired property of the program; then, there is no need to consider all possible fields in the program. The prototypical implementation of the abstract semantics includes a *field abstraction* mechanism which consists of aggregating all non-relevant fields into a special field "$\overline{any}$". Following the example of the double-linked list, the set φ can be abstracted to $\mathcal{F}' = \{next, prev, \overline{any}\}$, so that each path-formula has at most 64 models, and bit vectors only contain 6 bits. The analysis works on abstracted sets of fields by adding $\overline{any}$ to a model every time a path in the heap can traverse a field different from *next* and *prev*. Field abstraction can hugely improve scalability, and the loss of information is not relevant if it is clear which fields are actually important for the analysis. The prototypical implementation allows to specify manually which fields have to be represented by $\overline{any}$.

Example 5.1. *Consider the code* `x.f := y; z.g := y`, *and let* `f` *be the only relevant field: therefore, the field abstraction mechanism only considers* `f` *and the special field* $\overline{any}$. *The sharing information* $I_s(\mathsf{x}, \mathsf{z})$ *will be* $\left\langle \nwarrow_{\mathsf{f}}, \overline{any} \right\rangle$.

Finally, *program slicing* can be used to reduce the portion of code to be analyzed, if the sharing analysis will be used to assess some property at some program point: the slice will be the part of the program potentially affecting the property under study, and code outside the slice need not be analyzed.

6. An example

Consider the code fragment in Figure 10. The main method builds a tree and stores it into `w`; then it makes a shallow copy of the tree and stores it into `z`; finally, it swaps each left sub-tree and right sub-tree, recursively. Clearly, this algorithm has no practical meaning, but shows that the implementation can deal with method call, even of recursive methods.

After analyzing the main method, the implementation shows the sharing about (z, z) as a list of models like (L, P and R are shorthands for `left`, `parent` and `right`, respectively)

```
1  public class Tree {
2    Tree parent, left, right;
3    int data;
4
5    public Tree(int d,Tree l, Tree r) { // default constructor
6      data = d;  left = l;  right = r;
7      if (left!=null) left.parent = this;
8      if (right!=null) right.parent = this; }
9
10   public void mirror() { // ''mirroring'' the tree
11     if (left!=null) left.mirror();
12     if (right!=null) right.mirror();
13     Tree x = left;
14     left = right;
15     right = x; }
16
17   public Tree clone() { // shallow copy
18     return new Tree(data,left,right); }
19 }
20
21 public static void main(String[] args) {
22   Tree x = new Tree(4,null,null);
23   Tree y = new Tree(9,null,null);
24   Tree w = new Tree(6,x,y);
25   Tree z = w.clone();
26   z.mirror(); }
```

Figure 10: A tree with parent nodes and the main method

{}-{} (corresponding to $\neg \searrow_L \wedge \neg \searrow_R \wedge \neg \searrow_R \wedge \neg \swarrow_L \wedge \neg \swarrow_P \wedge \neg \swarrow_R$)

...

{left}-{left parent right} ($\searrow_L \wedge \neg \searrow_R \wedge \neg \searrow_R \wedge \swarrow_L \wedge \swarrow_P \wedge \swarrow_R$)

...

{right}-{right} ($\neg \searrow_L \wedge \neg \searrow_R \wedge \searrow_R \wedge \neg \swarrow_L \wedge \neg \swarrow_P \wedge \swarrow_R$)

...

{left parent right}-{left parent right} ($\searrow_L \wedge \searrow_R \wedge \searrow_R \wedge \swarrow_L \wedge \swarrow_P \wedge \swarrow_R$)

A total of 25 models is output by the analyzer, although the intended formula has 43 models: this is because, unlike the abstract semantics, the analyzer does take into account the symmetry of sharing, so that, for example, the model {left parent right}-{left} is not output because of {left}-{left parent right}.

The path-formula corresponding to these models can be written as

$$\begin{aligned}
\searrow_R &\rightarrow \searrow_L \vee \searrow_R && \wedge \quad (1)\\
\swarrow_P &\rightarrow \swarrow_L \vee \swarrow_R && \wedge \quad (2)\\
\neg \searrow_L \wedge \neg \searrow_R \wedge \neg \searrow_R &\rightarrow \swarrow_P \vee (\neg \swarrow_L \wedge \neg \swarrow_R) && \wedge \quad (3)\\
\neg \swarrow_L \wedge \neg \swarrow_P \wedge \neg \swarrow_R &\rightarrow \searrow_R \vee (\neg \searrow_L \wedge \neg \searrow_R) && \quad (4)
\end{aligned}$$

meaning that (1,2) no converging paths can simply "go up" in the tree: if paths traverse parent, then they also have to traverse left or right, or both; and (3,4) if one of the converging paths is empty, then the other cannot just "go down" in the tree: it must necessarily traverse parent (and also at least one between left or right because of (1) and (2)).

It is easy to see that this is a quite exhaustive (and sound) description of the sharing patterns about z. As a matter of fact, there are also some spurious models which should be considered as a loss of precision of the analysis: for example, {left}-{right} is a model of the formula but does not correspond to a real sharing pattern. This is due to the operation performed by mirror(), which exchanges the left and right sub-trees. In fact, the sharing information about w is more precise:

$$\begin{aligned}
(1) \wedge (2) \wedge (3) \wedge (4) &\quad \wedge\\
\searrow_L \wedge \swarrow_R \rightarrow (\swarrow_P \wedge \swarrow_L) \vee (\searrow_R \wedge \searrow_R) &\quad \wedge \quad (5)\\
\swarrow_L \wedge \searrow_R \rightarrow (\searrow_R \wedge \searrow_L) \vee (\swarrow_P \wedge \swarrow_R) &\quad \wedge \quad (6)\\
\searrow_L \wedge \searrow_R \rightarrow \searrow_R &\quad \wedge \quad (7)\\
\swarrow_L \wedge \swarrow_R \rightarrow \swarrow_P &\quad \wedge \quad (8)
\end{aligned}$$

meaning that (5,6) converging paths cannot traverse different sub-trees if one of them does not "go back" and traverse also the other subtree; and (7,8) no path can only traverse left and right without traversing parent. Conditions (7) and (8) come from the fact that the height of the tree stored in w is detected to be 1; note that this information is lost in the final value of z, due to the manipulations occurring in clone() and mirror.

7. Applications

This section discusses how field-sensitive sharing analysis can improve existing static analysis frameworks.

The closest, most related application of field-sensitive sharing analysis is the improvement of the corresponding field-sensitive *reachability/acyclicity* analysis [38]: there, one of the main sources of imprecision was the use of the traditional notion of sharing in some parts of the abstract semantics. Actually, in order to improve the precision w.r.t. the usual version of sharing, a related property of *deep sharing* was used in that analysis: v_1 and v_2 deep-share if there exist paths from both to a common location (as in sharing), and none of those paths is empty. Especially when method calls come into play, the use of field-sensitive sharing instead of deep-sharing and aliasing would lead to more precise results. For example, the computation of I_s'''' and Example 4.13 show that the new information about ρ is more precise than the *true* path-formula computed by the deep-sharing-based reachability analysis.

The example in the introduction suggests that to have field-sensitive information may help a number of static analyses like *termination* and *resource-usage* analysis. The computation of ranking functions allows to compute the number of iterations of a loop or, at least guarantee that an upper bound to that number exists. The use of a precise notion of sharing makes it possible to guarantee that important properties of a variable (or the corresponding data structure) are not broken by modifications to the heap, even if traditional sharing holds.

8. Conclusions

The analysis described in the present paper computes sharing information which includes information about the fields traversed by converging paths (i.e., paths corrisponding to sharing). This information is represented as boolean formulas which are more general than the most related approaches to sharing analysis.

This sound and precise information allows to guarantee, in some cases, that a modification to a variable does not affect another variable that is sharing with the first one in the traditional sense. Field-sensitive sharing information can be used to improve the inference of interesting sharing-dependent properties of heap-manipulating programs, such as (a)cyclicity of data structures or termination.

Ongoing and future work. The most obvious direction of ongoing and future work is to improve the implementation of the analysis: as discussed in Section 5, the current tool uses the "old" internal representation described in previous work [38], while the new representation is amenable to be translated into a more efficient BDD-based implementation. The recollection of experimental results is another direction of future work. Furthermore, effort will be devoted to the integration of the analysis into acyclicity [38], termination [21, 1] or resource-usage analyzers [21, 2].

Acknowledgments. This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006. Thanks to my former students Javier González Bodas and Miguel García Biedma for their help with technical issues and the implementation.

References

[1] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination Analysis of Java Bytecode. In *Procs. of the International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 2–18, 2008.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[3] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape Analysis by Predicate Abstraction. In *Procs. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 164–180, 2005.

[4] Sebastien Bardin, Alain Finkel, and Dadiv Nowak. Toward Symbolic Verification of Programs Handling Pointers. In *Procs. of International Workshop on Automated Verification of Infinite-State Systems (AVIS)*, 2004.

[5] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Procs. of International Conference on Computer Aided Verification (CAV)*, pages 386–400, 2006.

[6] Annalisa Bossi, , Maurizio Gabbrielli, Giorgio Levi, and Maurizio Martelli. The s-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19&20:149–197, 1994.

[7] Ahmed Bouajjani, Peter Habermehl, Pierre Moro, and Tomás Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Procs. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 13–29, 2005.

[8] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Procs. of Symposium on Static Analysis (SAS)*, pages 52–70, 2006.

[9] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In *Procs. of International Conference on Computer Aided Verification (CAV)*, pages 372–386, 2004.

[10] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated Termination Proofs for Java Programs with Cyclic Data. In *Procs. of International Conference on Computer Aided Verification (CAV)*, pages 105–122, 2012.

[11] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic Proofs of Program Termination in Separation Logic. In *Procs. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 101–112, 2008.

[12] Francisco Bueno and María García de la Banda. Set-Sharing Is Not Always Redundant for Pair-Sharing. In *Procs. of International Symposium on Functional and Logic Programming (FLOPS)*, pages 117–131, 2004.

[13] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Procs. of ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, 2006.

[14] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Procs. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[15] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *Procs. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[16] Samir Genaim and Fausto Spoto. Constancy Analysis. In *Procs. of Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2008.

[17] Samir Genaim and Damiano Zanardini. The acyclicity inference of COSTA. In *Procs. of International Workshop on Termination (WST)*, 2010.

[18] Samir Genaim and Damiano Zanardini. Reachability-based Acyclicity Analysis by Abstract Interpretation. *Theoretical Computer Science*, 474(0):60–79, 2013.

[19] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Procs. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–15, 1996.

[20] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Procs. of International Conference on Computer Aided Verification (CAV)*, pages 72–83, 1997.

[21] The COSTA Group. COSTA: COSt and Termination Analyzer for Java Bytecode. `http://costa.ls.fi.upm.es/~costa/costa/costa.php`.

[22] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Procs. of ACM Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[23] Mark Marron. Heap analysis design: An empirical approach. Technical Report MSR-TR-2014-46, 2014.

[24] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-Checking: A Tutorial Introduction. In *Procs. of Static Analysis Symposium (SAS)*, pages 330–354, 1999.

[25] Mayur Naik. *Chord: A Versatile Platform for Program Analysis*, 2011. User manual.

[26] Greg Nelson. Verifying Reachability Invariants of Linked Structures. In *Procs. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–47, 1983.

[27] Durica Nikolic and Fausto Spoto. Reachability analysis of program variables. *ACM Transaction on Programming Languages and Systems*, 35(4), 2013.

[28] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Procs. of IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[29] Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Procs. of ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

[30] Stefano Rossignoli and Fausto Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *Procs. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 95–110, 2006.

[31] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[32] Enrico Scapin and Fausto Spoto. Field-sensitive unreachability and non-cyclicity analysis. *Science of Computer Programming*, 95:359–375, 2014.

[33] Stefano Secci and Fausto Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Procs. of Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science. Springer, 2005.

[34] Fausto Spoto and Thomas P. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Transactions on Programming Languages and Systems*, 25(5):578–630, 2003.

[35] Fausto Spoto, Fred Mesnard, and Étienne Payet. A Termination Analyser for Java Bytecode based on Path-Length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[36] John Whaley. JavaBDD - Java Binary Decision Diagram library. `http://javabdd.sourceforge.net`.

[37] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Procs. of the International Conference on Compiler Construction (CC)*, 2000.

[38] Damiano Zanardini and Samir Genaim. Inference of Field-Sensitive Reachability and Cyclicity. *ACM Transactions on Computational Logic*, 15(4):33:1–33:41, 2014.