Scalable Shape Analysis For Systems Code

Hongseok Yang[1], Oukseh Lee[2], Josh Berdine[3], Cristiano Calcagno[4], Byron Cook[3], Dino Distefano[1], and Peter O'Hearn[1]

[1] Queen Mary, Univ. of London [2] Hanyang University, Korea
[3] Microsoft Research [4] Imperial College

Abstract. Pointer safety faults in device drivers are one of the leading causes of crashes in operating systems code. In principle, shape analysis tools can be used to prove the absence of this type of error. In practice, however, shape analysis is not used due to the unacceptable mixture of scalability and precision provided by existing tools. In this paper we report on a new join operation $\uplus$ for the separation domain which aggressively abstracts information for scalability yet does not lead to false error reports. $\uplus$ is a critical piece of a new shape analysis tool that provides an acceptable mixture of scalability and precision for industrial application. Experiments on whole Windows and Linux device drivers (firewire, pci-driver, cdrom, md, etc.) represent the first working application of shape analysis to verification of whole industrial programs.

1 Introduction

Pointer safety faults in device drivers are one of the leading causes of operating system crashes. The reasons for this are as follows:

- The average Windows or Linux computer has numerous (*i.e.* >15) device drivers installed,
- Most device drivers manage relatively complex combinations of shared singly- and doubly-linked lists,
- Device drivers are required to respect many byzantine invariants while manipulating data structures (*e.g.* pieces of data structures that have been paged out can only be referenced at low thread-priority). This results in complex and nonuniform calling conventions, unlike typical benchmark code.

By pointer safety we mean that a program does not dereference null or a dangling pointer, or produce a memory leak. In principle a shape analysis tool can be used to prove the absence of pointer safety violations: shape analysis is a heap-aware program analysis with accurate handling of deep update. Furthermore, device drivers are small (*e.g.* <15k LOC) and usually do not use trees or DAGs—thus making device drivers the perfect application for shape analysis.

So, why aren't shape analysis tools regularly applied to device drivers? The reason is that today's shape analysis tools are either scalable, or precise, but not both. Numerous papers have reported on the application of accurate shape analysis to small examples drawn from real systems code; other papers have

Program	LOC	Sec	MB	Memory leaks	Dereference errors	False error rate
`scull.c`	1010	0.36	0.25	1	0	0%
`class.c`	1983	8.21	7.62	2	1	0%
`pci-driver.c`	2532	0.97	1.72	0	0	0%
`ll_rw_blk.c`	5469	887.94	485.87	3	1	0%
`cdrom.c`	6218	103.26	71.52	0	2	0%
`md.c`	6635	1585.69	847.63	6	5	0%
`t1394Diag.c`	10240	135.05	68.81	33	10	0%

Table 1. Results with the ⧢ extension of SpaceInvader on Windows and Linux device drivers. Experiments were performed on an Intel Core Duo 2.0GHz with 2GB. Each error found was confirmed manually. Errors in the Windows device driver (`t1394Diag.c`) were confirmed by the Windows kernel team. The time and space columns contain the numbers for the analysis of fixed versions of the drivers (and so report time to find proofs of pointer safety).

reported on very imprecise analysis on large code bases. The verification of whole industrial programs, however, requires both.

Towards the elusive goal of finding a scalable *and* precise analysis, in this paper we describe a new join operation, ⧢, for shape analysis tools based on the separation domain [10, 17, 4]. ⧢ provides a mixture of scalability and precision sufficient for the problem of proving pointer safety of whole industrial device drivers. A join operation (in the terminology of abstract interpretation [9]) takes a disjunction of two abstract states, each of which describes (in our setting) a set of concrete heaps that may arise during program execution. ⧢ attempts to construct a common generalization of the states. In case the attempt succeeds (⧢ is a *partial* join operator) the generalization subsequently replaces the original disjunction, leading to fewer cases to consider during the shape analysis.

In order to demonstrate the scalability and accuracy of ⧢, we have implemented it in our shape analysis tool SpaceInvader, together with an abstract model of the operating system environment that we have developed. Then, we have applied the resulting tool to numerous Windows and Linux device drivers.

2 Experiments

Before describing the technical details of ⧢, we first present the results of an experimental evaluation that demonstrates its scalability and precision. Table 1 displays the results of experiments with the ⧢ extension of SpaceInvader on seven device drivers. Each of the drivers manipulates multiple, sometimes nested, sometimes circular, linked lists. One driver, `t1394Diag.c`, is the IEEE 1394 (firewire) driver for the Windows operating system. The drivers `pci-driver.c`, `ll_rw_blk.c`, `cdrom.c` and `md.c` are from an industrial version of Embedded Linux, given to us by ETRI. The driver `class.c` is from a standard Linux dis-

tribution, and `scull.c` is a Linux char driver used in the experiments in [7].[1] Each of these drivers is analyzed in the context of environment code which nondeterministically generates input data structures, and calls the driver's dispatch routines repeatedly. In essence, each driver is supplied with a particular precondition (expressed as C code, as in [7]) but the model of system calls can be reused from driver to driver.

During our experiments SpaceInvader was used in a stop-first configuration, where the analyzer halts if it cannot prove that a dereferencing operator is safe or if it cannot prove that a cell is reachable. When we encountered bugs we would fix them, and then run our tool again. The time and space columns in Table 1 report the numbers for the analysis of our bug-fixed versions of the drivers. Note that, during our experiments, *no false errors were found*. Also, note that for the fixed drivers SpaceInvader *proved* pointer safety. No known tool with scalability reported to programs up to 10k LOC can match that precision.

Caveats. Device drivers often use circular doubly-linked lists. The first caveat is that, in several cases, we modified the examples in order to operate over singly-linked lists, in order to aid our analysis. Pointer safety can often be proved using singly-linked semantics even though the code is designed to operate over doubly-linked lists (it is rare for code to actually make use of the back pointers). Second, there is a significant caveat regarding arrays. SpaceInvader currently presumes memory safety of arrays, by returning a nondeterministic value for any array dereferencing. The treatment of pointer safety can still be sound under such an assumption, and in the (slightly modified) Linux drivers our analyzer encountered no false alarms. However, the 1394 device driver contains arrays of pointers, which are beyond what our method can handle: we modified the code such that those arrays have size 1 and can be treated as pointer variables. This, of course, is just one instance of the fact that the problems of analyzing arrays and pointers are not independent. We regard this issue as an avenue for interesting future work. Finally, note that SpaceInvader currently only implements shape analysis for sequential programs, whereas device drivers of course are multi-threaded. As reported in [12], a sequential shape analysis tool such as SpaceInvader can be used to find and then verify resource invariants for device drivers, thereby proving pointer safety for the concurrent program. However, we emphasize that developing a scalable, precise shape analysis for concurrent programs is an open problem; only very recently, some interesting ideas such as [12, 18, 5] have been proposed, which give promising new lines of attack, but on which further, especially experimental, work is needed.

3 Abstract States and Setting

In this section we describe the abstract states that SpaceInvader analysis operates over. In the next section we will describe the details of ⫫. Due to space

[1] This is a modified version of the Linux scull driver, where arrays are assumed to be of size 1.

constraints we will assume that the reader is somewhat familiar with the basics of program analysis and shape analysis.

SpaceInvader operates over abstract states expressed as separation logic formulae. Following [4, 10, 17], we call these abstract states *symbolic heaps*. The symbolic heaps q, are defined by the following grammar:

$$\begin{array}{ll} e ::= x \mid x' \mid 0 & \mathcal{P} ::= \cdots \\ \Pi ::= \Pi \wedge \Pi \mid e{=}e \mid e{\neq}e \mid \mathsf{true} & \Sigma ::= \Sigma * \Sigma \mid \mathsf{emp} \mid \mathcal{P} \mid \mathsf{true} \\ q ::= \mathsf{err} \mid \Pi \wedge \Sigma & \end{array}$$

A symbolic heap q can be err, denoting the error state, or it has the form $\Pi \wedge \Sigma$, where Π and Σ describe properties of variables and the heap, respectively. The separating conjunction $\Sigma_0 * \Sigma_1$ holds for a heap if and only if the heap can be split into two disjoint parts, one making Σ_0 true and the other making Σ_1 true. emp means the empty heap, and true holds for all heaps. The primed variables x' in a symbolic heap are assumed to be (implicitly) existentially quantified.

$\mathcal{P}$ is a collection of basic predicates. One instantiation is

$$k \ ::= \ \mathsf{PE} \mid \mathsf{NE} \qquad \mathcal{P} \ ::= \ (e \mapsto e) \mid \mathsf{ls}\, k\, e\, e$$

Here, $e \mapsto f$ means a heap with only one cell e that stores f. The list segment predicate $\mathsf{ls}\, k\, e_0\, e_1$ denotes heaps containing one list segment from e_0 to e_1 only. This list segment starts at cell e_0 and its last cell stores e_1. The list is possibly empty if $k = \mathsf{PE}$; otherwise (*i.e.*, $k = \mathsf{NE}$), the list is not empty. The meanings of the segment predicates can be understood in terms of the definitions

$$\begin{array}{l} \mathsf{ls}\,\mathsf{PE}\, e\, f \iff (e = f \wedge \mathsf{emp}) \vee (\mathsf{ls}\,\mathsf{NE}\, e\, f), \\ \mathsf{ls}\,\mathsf{NE}\, e\, f \iff (e \mapsto f) \vee (\exists y'.\, e \mapsto y' * \mathsf{ls}\,\mathsf{NE}\, y'\, f). \end{array}$$

These definitions are not *within* the shape domain (*e.g.*, the domain does not have $\vee$), but are mathematical definitions in the metalanguage, used to verify soundness of operations on the predicates. Note that there is no problem with the recursion in $\mathsf{ls}\,\mathsf{NE}$: the recursive instance is in a positive position, and the definition satisfies monotonicity properties sufficient to ensure a solution.

A different instantiation of $\mathcal{P}$ gives us a variation on [3].[2]

$$k \ ::= \ \mathsf{PE} \mid \mathsf{NE} \qquad \mathcal{P} \ ::= \ (e \mapsto \vec{\mathtt{f}} : \vec{e}) \mid \mathsf{ls}\, k\, \phi\, e\, e$$

Here, the points-to predicate $(e \mapsto \vec{\mathtt{f}} : \vec{e})$ is for records with fields $\vec{\mathtt{f}}$, and ϕ is a binary predicate that describes the shape of each node in a list. The definition of the nonempty list segment here is

$$\mathsf{ls}\,\mathsf{NE}\, \phi\, e\, f \iff \phi(e, f) \vee (\exists y'.\, \phi(e, y') * \mathsf{ls}\,\mathsf{NE}\, y'\, f)$$

and the ϕ predicate gives us a way to describe composite structures.

[2] This instantiation assumes the change of the language where we have heap cells with multiple fields, instead of unary cells.

For example, if $\mathtt{f}$ is a field, let $\phi_{\mathtt{f}}$ be the predicate where $\phi_{\mathtt{f}}(x, y)$ is $x \mapsto \mathtt{f}\colon y$. Then using $\phi_{\mathtt{f}}$ as ϕ, the formula $\mathsf{ls}\,\mathsf{NE}\,\phi\,e\,f$ describes lists linked by the $\mathtt{f}$ field. The formula

$$(x \mapsto \mathtt{f}\colon y', \mathtt{g}\colon z') \;*\; \mathsf{ls}\,\mathsf{PE}\,\phi_{\mathtt{f}}\,y'\,x \;*\; \mathsf{ls}\,\mathsf{PE}\,\phi_{\mathtt{g}}\,z'\,x$$

describes two circular linked lists sharing a common header, where one list uses $\mathtt{f}$ for linking and the other uses $\mathtt{g}$. Finally, if ϕ itself describes lists, as when $\phi(x, y)$ is the predicate $\exists x'.\,(x \mapsto \mathtt{g}\colon x', \mathtt{f}\colon y) * \mathsf{ls}\,\mathsf{PE}\,\phi_{\mathtt{g}}\,x'\,0$, then $\mathsf{ls}\,\mathsf{NE}\,\phi\,e\,f$ describes a nonempty linked list where each node points to a possibly empty sublist, and where the sublists are disjoint. Combinations of these kinds of structures, nested lists and multiple lists with a common header node, are common in device drivers.

The experiments in this paper are done using this second instantiation of $\mathcal{P}$. It is similar to the domain from [3], but uses predicates for both possibly empty and necessarily nonempty list segments. The reader might have noticed that having $\mathsf{ls}\,\mathsf{PE}$ does not give us any extra expressive power: its meaning can be represented using two abstract states, one a emp and the other a $\mathsf{ls}\,\mathsf{NE}$. However, having $\mathsf{ls}\,\mathsf{PE}$ impacts performance, as it represents disjunctive information, succinctly.

SpaceInvader implements a context sensitive, flow sensitive analysis, using a variant of the RHS interprocedural dataflow analysis algorithm [22, 11]. It employs join to make procedure summaries smaller. Following [21, 23], SpaceInvader also passes only the reachable portion of the heap to a procedure and aggressively discards intermediate states. The mixture of these optimizations—join, locality, discarding states—is key; turning off any one of the optimizations results in the analysis using more than the 2GB RAM on at least one of the examples, causing disk thrashing, and then leading to timeout (which we set at 90min). Thus we do not claim that $\Cup$ alone is the root cause for the performance found in Table 1, but it is a critical ingredient (c.f., §4.3).

4 A Join for Symbolic Heaps

We now discuss $\Cup$. We begin with an intuitive explanation. Later, in §4.1, we provide a formal definition.

In the framework of abstract interpretation [9], a join operator takes two symbolic states in a program analysis and attempts to find a common generalization. To see the issue, consider the program

```
x=0;  while (NONDET) { d=malloc(sizeof(Node)); d->next=x; x=d; }
```

which nondeterministically generates acyclic linked lists. When we run our basic analysis algorithm, without $\Cup$, it returns three symbolic heaps at the end: $(\mathsf{ls}\,\mathsf{NE}\,x\,0) \vee (x \mapsto 0) \vee (x = 0 \wedge \mathsf{emp})$. (Here, for simplicity in the presentation, we have elided the ϕ parameter of the ls predicates.)

Now, if you look at the first two disjuncts there is evident redundancy: If you know that either x points to 0 or a nonempty linked list, then that is the same as knowing you have a nonempty linked list. So, $\Cup$ replaces the first two

Program	NO JOIN NE	NO JOIN PE	JOIN NE	JOIN PE
`onelist_create.c`	3	3	2	1
`twolist_create.c`	9	9	4	1
`firewire_create.c`	3969	3087	32	1

Table 2. Creation routines. Reports the number of states in the postcondition with join turned on or off, and the base list predicates chosen to be either nonempty ls only (NE), or both nonempty and possibly empty ls (PE).

disjuncts with just the list segment formula, giving us $(\mathsf{ls}^{\mathsf{NE}}\, x\, 0) \vee (x = 0 \wedge \mathsf{emp})$. It is possible to take yet a further step, using the notion of a possibly empty list segment. If you know that either you have a nonempty list, or that $x = 0 \wedge \mathsf{emp}$, then that is the same as having $\mathsf{ls}^{\mathsf{PE}}\, x\, 0$, and ⧺ produces this formula from the previous two. Thus, using ⧺ we have gone from a position where we have three disjuncts in our postcondition, to where we have only one. The saving that this possibly gives us is substantial, especially for more complicated programs or more complicated data structures.

Table 2 gives an indication. `onelist_create.c` in the table is the C program above that nondeterministically creates a list and `twolist_create.c` is a similar C program that creates two disjoint linked lists. `firewire_create.c` is the environment code we use in the analysis of the 1394 firewire driver: it creates five cyclic linked lists, which share a common header node, with head pointers in some of the lists, and with nested sublists.

There are two points to note. The first is just the great saving, in number of states (*e.g.*, from 3087 down to 1). This is particularly important with environment code, like `firewire_create.c`, which is run as a harness to generate heaps on which driver routines will subsequently be run. The second is the distinction between NE and PE. In the table we keep track of two versions of our analysis, one where $\mathsf{ls}^{\mathsf{NE}}$ is the only list predicate used by the analysis, and another where we use both $\mathsf{ls}^{\mathsf{NE}}$ and $\mathsf{ls}^{\mathsf{PE}}$.

This illustration shows some of the aspects of ⧺, but not all. In the illustration ⧺ worked perfectly, never losing any information, but this is not always the case. Part of the intuition is that you generalize points-to facts by list segments when you can. So, considering $y \mapsto 0 * (\mathsf{ls}^{\mathsf{NE}}\, x\, 0) \vee (\mathsf{ls}^{\mathsf{NE}}\, y\, 0) * x \mapsto 0$, ⧺ will produce $(\mathsf{ls}^{\mathsf{NE}}\, y\, 0) * (\mathsf{ls}^{\mathsf{NE}}\, x\, 0)$. This formula is less precise than the disjunction, in that it loses the information that one or the other of the lists pointed to by x and y has length precisely 1. Fortunately, it is unusual for programs to rely on this sort of disjunctive information.

We have tried to keep the intuitive description simple, but the truth is that ⧺ must deal with disequalities, equalities, and generalization of "nothing" by $\mathsf{ls}^{\mathsf{PE}}$ in ways that are nontrivial. It also must deal with the existential (primed)

variables specially. In the end, for instance, when $\natural$ is given

$$\begin{array}{rcl} q_0 & \equiv & x{\neq}y \wedge (\mathsf{ls}_{\mathsf{NE}}\, x\, 0 * y \mapsto 0) \qquad \text{and} \\ q_1 & \equiv & x{\neq}y \wedge x'{\neq}y \wedge (x \mapsto x' * \mathsf{ls}_{\mathsf{NE}}\, y\, x' * \mathsf{ls}_{\mathsf{NE}}\, x'\, 0), \end{array}$$

it will produce $x{\neq}y \wedge \mathsf{ls}_{\mathsf{NE}}\, x\, v' * \mathsf{ls}_{\mathsf{NE}}\, y\, v' * \mathsf{ls}_{\mathsf{PE}}\, v'\, 0$. Now we turn to the formal definition.

4.1 Formal Definition

In this section, we define the (partial) binary operator $\natural$ on symbolic heaps, considering only the simple linked lists (the first instantiation of $\mathcal{P}$). The $\natural$ for nested lists will be described in the next section.

$\natural$ works in two stages. Suppose that it is given symbolic heaps $(\Pi_0 \wedge \Sigma_0)$ and $(\Pi_1 \wedge \Sigma_1)$ that do not share any primed variables. In the first stage, $\natural$ constructs Σ and a ternary relation ϵ' on expressions such that

$$(1) \qquad \forall i \in \{0,1\}.\ \left(\bigwedge\{e_i{=}x' \mid (e_0, e_1, x') \in \epsilon'\}\right) \wedge \Sigma_i \implies \Sigma.$$

Intuitively, this condition means that Σ overapproximates both Σ_0 and Σ_1, and that ϵ' provides witnesses of existential (primed) variables of Σ for this overapproximation. For instance, if $\Sigma_0 \equiv (\mathsf{ls}_{\mathsf{NE}}\, x\, 0 * y \mapsto 0)$ and $\Sigma_1 \equiv (x \mapsto x' * \mathsf{ls}_{\mathsf{NE}}\, y\, x' * \mathsf{ls}_{\mathsf{NE}}\, x'\, 0)$, then $\natural$ returns

$$(2) \qquad \Sigma \equiv \mathsf{ls}_{\mathsf{NE}}\, x\, v' * \mathsf{ls}_{\mathsf{NE}}\, y\, v' * \mathsf{ls}_{\mathsf{PE}}\, v'\, 0, \qquad \epsilon' \equiv \{(0, x', v')\}.$$

In this case, the condition (1) is

$$\begin{array}{rcl} 0{=}v' \wedge (\mathsf{ls}_{\mathsf{NE}}\, x\, 0 * y \mapsto 0) & \implies & (\mathsf{ls}_{\mathsf{NE}}\, x\, v' * \mathsf{ls}_{\mathsf{NE}}\, y\, v' * \mathsf{ls}_{\mathsf{PE}}\, v'\, 0) \\ x'{=}v' \wedge (x \mapsto x' * \mathsf{ls}_{\mathsf{NE}}\, y\, x' * \mathsf{ls}_{\mathsf{NE}}\, x'\, 0) & \implies & (\mathsf{ls}_{\mathsf{NE}}\, x\, v' * \mathsf{ls}_{\mathsf{NE}}\, y\, v' * \mathsf{ls}_{\mathsf{PE}}\, v'\, 0). \end{array}$$

This means that both Σ_0 and Σ_1 imply Σ when 0 and x' are used as witnesses for the (implicitly) existentially quantified variable v' of Σ.

After constructing Σ and ϵ', the $\natural$ operator does one syntactic check on ϵ', in order to decide whether it has lost crucial sharing information of input symbolic heaps. Only when the check succeeds does $\natural$ move on to the second stage. (We will describe the details of the first stage, including the check on ϵ', later.)

In the second stage, the $\natural$ operator computes an overapproximation Π of Π_0 and Π_1:

$$\Pi \stackrel{def}{=} \bigwedge \begin{pmatrix} \{e{=}f \mid e{=}f \text{ has no primed vars, it occurs in } \Pi_0 \text{ and } \Pi_1\} \\ \cup\, \{e{\neq}f \mid e{\neq}f \text{ has no primed vars, it occurs in } \Pi_0 \text{ and } \Pi_1\} \\ \cup\, \{x'{\neq}0 \mid (e_0, e_1, x') \in \epsilon' \text{ and } e_i{\neq}0 \text{ occurs in } \Pi_i\} \end{pmatrix}.$$

This definition says that $\natural$ keeps an equality or disequality in Π if it appears in both Π_0 and Π_1 and does not contain any primed variables, or if it is of the form $x'{\neq}0$ and its witness e_i for the i-th symbolic heap is guaranteed to be different from 0. Both cases are considered here in order to deal with programming

patterns found in device drivers. For instance, $x' \neq 0$ in the second case should be included, because some drivers store 0 or 1 to a cell, say, x, depending on whether a linked list y is empty, and subsequently, they use the contents of cell x to decide the emptiness of the list y. The computed Π and the result Σ of the first stage become the output of $\uplus$.

Computation of Σ, ϵ': We now describe the details of the first stage of $\uplus$. For this, we need a judgment

$$\Sigma_0, \Sigma_1, \epsilon \rightsquigarrow \Sigma, \epsilon', \delta_0, \delta_1$$

where δ_i is a binary relation on expressions in Σ_i. This judgment signifies that Σ_0 and Σ_1 can be joined to give Σ and a ternary relation ϵ' for witnesses. Furthermore, the judgment ensures that ϵ' extends the given ϵ, and that δ_i records (e_i, f_i) of all $\mathsf{ls}\, k\, e_i\, f_i$ in Σ_i that have been generalized to a possibly empty list during the join; these δ_i components are used later to decide whether this join to Σ has lost too much information and should, therefore, be discarded. For instance, we have

$$\begin{array}{rl} (\mathsf{ls}\,\mathsf{NE}\, x\, 0 * y \mapsto 0), & (x \mapsto x' * \mathsf{ls}\,\mathsf{NE}\, y\, x' * \mathsf{ls}\,\mathsf{NE}\, x'\, 0), \;\; \emptyset \\ \rightsquigarrow & (\mathsf{ls}\,\mathsf{NE}\, x\, v' * \mathsf{ls}\,\mathsf{NE}\, y\, v' * \mathsf{ls}\,\mathsf{PE}\, v'\, 0), \; \{(0, x', v')\}, \; \emptyset, \; \{(x', 0)\}. \end{array}$$

which means that $\Sigma_0 \equiv (\mathsf{ls}\,\mathsf{NE}\, x\, 0 * y \mapsto 0)$ and $\Sigma_1 \equiv (x \mapsto x' * \mathsf{ls}\,\mathsf{NE}\, y\, x' * \mathsf{ls}\,\mathsf{NE}\, x'\, 0)$ are joined to $\Sigma \equiv (\mathsf{ls}\,\mathsf{NE}\, x\, v' * \mathsf{ls}\,\mathsf{NE}\, y\, v' * \mathsf{ls}\,\mathsf{PE}\, v'\, 0)$. The judgment also says that v' in Σ corresponds to 0 in Σ_0 and x' in Σ_1. Note that the δ_1 component of the judgment is $\{(x', 0)\}$, and it reflects the fact that $\mathsf{ls}\,\mathsf{NE}\, x'\, 0$ in Σ_1 is generalized to a possibly empty list and results in $\mathsf{ls}\,\mathsf{PE}\, v'\, 0$ in Σ.

The derivation rules of the $\rightsquigarrow$ predicate are given in Figure 1. The first two rules deal with the cases when emp or true appear in both Σ_0 and Σ_1. The third rule has to do with generalizing two lists or abstracting a points-to to a list, and the last two rules are about generalizing (or synthesizing) possibly empty lists. Note that when possibly empty lists are introduced by the last two rules, the appropriate δ_i component is extended with the information about the ls predicate of Σ_i that supports this generalization.

The first stage of $\uplus$ works as follows:

1. $\uplus$ searches for $\Sigma, \epsilon', \delta_0, \delta_1$ for which $\Sigma_0, \Sigma_1, \emptyset \rightsquigarrow \Sigma, \epsilon', \delta_0, \delta_1$ can be derived using the rules in Figure 1. This proof search proceeds by viewing rules backward from conclusion to premise. It searches for a rule whose conclusion has the left hand side matching with $\Sigma_0, \Sigma_1, \epsilon$ and whose side condition is satisfied with this matching. Once such a rule is found, the search modifies $\Sigma_0, \Sigma_1, \epsilon$ such that they fit the left hand side of the $\rightsquigarrow$ judgment in the premise. The search continues with this modified $\Sigma_0, \Sigma_1, \epsilon$, until it hits the base case (*i.e.*, the first rule in Figure 1). Figure 2 shows an example proof search. If the search fails, the join fails.
2. $\uplus$ checks whether for all $(e_0, e_1, e), (f_0, f_1, f) \in \epsilon' \cup \{(e, e, e) \mid e \text{ not primed var}\}$ and all $i \in \{0, 1\}$,

$$\big(e_i = f_i \;\wedge\; e_i \neq 0 \implies (e_{1-i}, f_{1-i}) \in \mathsf{eq}(\delta_{1-i})\big),$$

$$A(e,f) \;::=\; (e \mapsto f) \mid \mathsf{ls}\, k\, e\, f \qquad \mathsf{EQ} = \{(e,e,e) \mid e \text{ is not a primed var}\}$$

$$\mathsf{PE} \sqcup \mathsf{NE} = \mathsf{NE} \sqcup \mathsf{PE} = \mathsf{PE} \sqcup \mathsf{PE} = \mathsf{PE} \qquad \mathsf{NE} \sqcup \mathsf{NE} = \mathsf{NE}$$

$$A(e,f) \sqcup A(e,f) = A(e,f) \quad (\mathsf{ls}\, k_0\, e\, f) \sqcup (\mathsf{ls}\, k_1\, e\, f) = (\mathsf{ls}\, (k_0 \sqcup k_1)\, e\, f)$$
$$(e \mapsto f) \sqcup (\mathsf{ls}\, k\, e\, f) = (\mathsf{ls}\, k\, e\, f) \sqcup (e \mapsto f) = \mathsf{ls}\, k\, e\, f$$

$$\frac{}{\mathsf{emp},\ \mathsf{emp},\ \epsilon \rightsquigarrow \mathsf{emp},\ \epsilon,\ \emptyset,\ \emptyset}\ \mathsf{emp} \qquad \frac{\Sigma_0\ ,\ \Sigma_1\ ,\ \epsilon \rightsquigarrow \Sigma\ ,\ \epsilon'\ ,\ \delta_0\ ,\ \delta_1}{\mathsf{true} * \Sigma_0,\ \mathsf{true} * \Sigma_1,\ \epsilon \rightsquigarrow \mathsf{true} * \Sigma,\ \epsilon',\ \delta_0,\ \delta_1}\ \mathsf{true}$$

$$\frac{\Sigma_0\ ,\ \Sigma_1\ ,\ \mathsf{ext}(\epsilon, f_0, f_1, f) \rightsquigarrow \Sigma\ ,\ \epsilon'\ ,\ \delta_0\ ,\ \delta_1}{A_0(e_0,f_0) * \Sigma_0\ ,\ A_1(e_1,f_1) * \Sigma_1\ ,\ \epsilon \rightsquigarrow (A_0(e,f) \sqcup A_1(e,f)) * \Sigma\ ,\ \epsilon'\ ,\ \delta_0\ ,\ \delta_1}\ \mathsf{match}$$
$$(\text{when } (e_0,e_1,e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ \mathsf{comb}_\epsilon(f_0,f_1) = f)$$

$$\frac{\Sigma_0\ ,\ \Sigma_1\ ,\ \mathsf{ext}(\epsilon, f_0, e_1, f) \rightsquigarrow \Sigma\ ,\ \epsilon'\ ,\ \delta_0\ ,\ \delta_1}{(\mathsf{ls}\, k\, e_0\, f_0) * \Sigma_0\ ,\ \Sigma_1\ ,\ \epsilon \rightsquigarrow (\mathsf{ls}\, \mathsf{PE}\, e\, f) * \Sigma\ ,\ \epsilon'\ ,\ \delta_0 \cup (e_0,f_0)\ ,\ \delta_1}\ \mathsf{PE\text{-}left}$$
$$(\text{when } (e_0,e_1,e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ e_1 \notin \mathsf{MayAlloc}(\Sigma_1) \ \wedge\ \mathsf{comb}_\epsilon(f_0,e_1) = f)$$

$$\frac{\Sigma_0\ ,\ \Sigma_1\ ,\ \mathsf{ext}(\epsilon, e_0, f_1, f) \rightsquigarrow \Sigma\ ,\ \epsilon'\ ,\ \delta_0\ ,\ \delta_1}{\Sigma_0\ ,\ (\mathsf{ls}\, k\, e_1\, f_1) * \Sigma_1\ ,\ \epsilon \rightsquigarrow (\mathsf{ls}\, \mathsf{PE}\, e\, f) * \Sigma\ ,\ \epsilon'\ ,\ \delta_0\ ,\ \delta_1 \cup (e_1,f_1)}\ \mathsf{PE\text{-}right}$$
$$(\text{when } (e_0,e_1,e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ e_0 \notin \mathsf{MayAlloc}(\Sigma_0) \ \wedge\ \mathsf{comb}_\epsilon(e_0,f_1) = f)$$

Here (a) we write $X \cup x$ instead of $X \cup \{x\}$; (b) $\mathsf{ext}(\epsilon, e_0, e_1, e)$ is $(\epsilon \cup (e_0,e_1,e)) - \mathsf{EQ}$; (c) $\mathsf{MayAlloc}(\Sigma)$ is the set of expressions that appear on the left hand side of a points-to predicate or as a first expression argument of ls in Σ; (d) comb_ϵ is a function defined as:

$$\mathsf{comb}_\epsilon(e_0,e_1) = \begin{cases} e & \text{if } (e_0,e_1,e) \in \epsilon \text{ for some } e \\ e_0 & \text{if } e_0{=}e_1 \text{ and } e_0 \text{ is not a primed var} \\ x' \text{ for some } x' \notin \mathsf{FV}(\epsilon,e_0,e_1) & \text{otherwise} \end{cases}$$

Fig. 1. Rules for $\rightsquigarrow$

where $\mathsf{eq}(\delta_i)$ is the least equivalence relation containing δ_i. Intuitively this condition amounts to the following: consider Σ_0 and Σ_1 viewed as graphs with edges for $\mapsto$ and ls, and then identify vertices according to the returned δ's, then they should be isomorphic via $\epsilon' \cup \{(e,e,e) \mid e \text{ not primed var}\}$. Only when the check succeeds does the first stage of $\Cup$ return Σ, ϵ'. For instance, given $\Sigma_0 \equiv (x \mapsto y) * \mathsf{ls}\, \mathsf{NE}\, y\, 0$ and $\Sigma_1 \equiv \mathsf{ls}\, \mathsf{NE}\, x\, 0 * (y \mapsto 0)$, the proof search in the previous step succeeds with

$$\Sigma \equiv \mathsf{ls}\, \mathsf{NE}\, x\, y' * \mathsf{ls}\, \mathsf{NE}\, y\, 0, \qquad \epsilon' \equiv \{(y,0,y')\}, \qquad \delta_0 \equiv \delta_1 \equiv \emptyset.$$

However, the final check on ϵ' fails, since y in the Σ_0 symbolic heap is related to both 0 (by ϵ') and y (by default) in Σ_1. Thus, the join fails. Note that the failure is desired in this case since Σ_0 and Σ_1 describe heaps with different shapes.

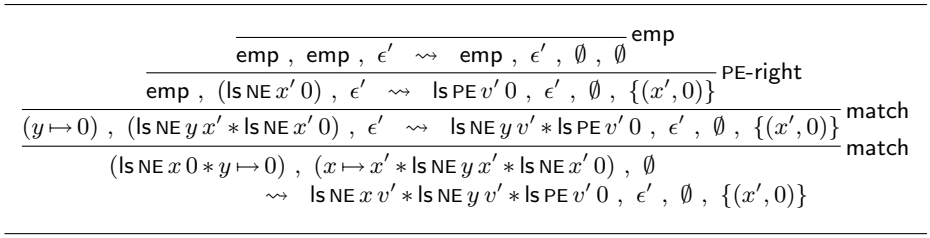

Fig. 2. Example proof search, where $\epsilon' = \{(0, x', v')\}$

4.2 Composite Structures

In order to handle composite structures, such as nested lists, we adjust the definition of ⧺ in the previous section. Specifically, we change the rules for the $\rightsquigarrow$ relation in Figure 1. Firstly, we modify the third rule, which is used to generalize two ls or points-to predicates, such that it can deal with points-to predicates with multiple fields $\vec{\mathtt{f}}$ and a parameterized list-segment predicate. Each of the new rules, shown in Figure 3, corresponds to one of the four cases of $A_0 \sqcup A_1$ in the third rule of Figure 1. The first rule combines two points-to predicates with multiple fields, by extending ϵ with the targets of all the fields. The other rules generalize two list-segment predicates (the second rule) or a list segment and its length-one instance (the third and fourth rules), by looking inside the two available descriptions of list nodes (denoted ϕ_0 and ϕ_1), and chooses the more general one (denoted $\phi_0 \sqcup \phi_1$). In the third rule of Figure 3, the first input symbolic heap is decomposed into $\phi_0(e_0, f_0)[\vec{e}/\vec{x'}] * \Sigma_0$ using a frame inference algorithm [4] to subtract a symbolic heap $\phi_0(e_0, f_0)[\vec{e}/\vec{x'}]$ such that ϕ_0 can be $\sqcup$-joined with ϕ_1, leaving Σ_0 as a remainder. And similarly in the fourth rule. Secondly, we change the remaining rules in Figure 1 such that they work with parameterized list-segment predicates. We simply replace all unparameterized list-segment predicates $\mathsf{ls}\,k\,e\,e'$ in the rules by parameterized ones $\mathsf{ls}\,k\,\phi\,e\,e'$.

After these changes, ⧺ works for composite structures. For instance, let $\phi_{\mathtt{d}}(x, y) \equiv (x \mapsto \mathtt{d}{:}y)$, $\phi(x, y) \equiv \exists x'.\,(x \mapsto \mathtt{d}{:}x', \mathtt{f}{:}y) * (\mathsf{ls}\,\mathsf{PE}\,\phi_{\mathtt{d}}\,x'\,0)$, and $\psi(x, y) \equiv (x \mapsto \mathtt{d}{:}0, \mathtt{f}{:}y)$. Given two symbolic heaps

$$(\mathsf{ls}\,\mathsf{NE}\,\phi\,x\,y) * (y \mapsto \mathtt{d}{:}y', \mathtt{f}{:}0) * (y' \mapsto \mathtt{d}{:}0) \quad \vee \quad (\mathsf{ls}\,\mathsf{PE}\,\psi\,x\,y) * (\mathsf{ls}\,\mathsf{PE}\,\phi\,y\,0),$$

the ⧺ generalizes the list segments from x to y to a possibly empty ϕ-shaped list since $\psi(x, y) \vdash \phi(x, y)$. Then, it views the two points-to facts on y and y' as an instantiation $\phi'(x, y)[y'/x']$ of $\phi'(x, y) \equiv \exists x'.\,(x \mapsto \mathtt{d}{:}x', \mathtt{f}{:}y) * (x' \mapsto \mathtt{d}{:}0)$, combines these facts with the list y since $\phi'(x, y) \vdash \phi(x, y)$, and produces

$$\mathsf{ls}\,\mathsf{PE}\,\phi\,x\,y \;*\; \mathsf{ls}\,\mathsf{PE}\,\phi\,y\,0.$$

4.3 Incorporating ⧺ into the analysis

SpaceInvader incorporates ⧺ together with RHS [22], a now-standard interprocedural analysis algorithm. RHS associates a set of symbolic heaps with each

$$\phi_0 \sqcup \phi_1 = \begin{cases} \phi_0 & \text{if } \phi_1(x,y) \vdash \phi_0(x,y) \\ \phi_1 & \text{if } \phi_0(x,y) \vdash \phi_1(x,y) \\ \text{undefined} & \text{otherwise} \end{cases} \quad \begin{array}{l} \text{where } \phi_0(x,y) \vdash \phi_1(x,y) \text{ denotes} \\ \text{a call to a sound theorem prover} \\ \text{for fresh } x, y \end{array}$$

$$\frac{\Sigma_0, \Sigma_1, \mathsf{ext}(\mathsf{ext}(\epsilon, f_0, f_1, f), g_0, g_1, g) \rightsquigarrow \Sigma, \epsilon', \delta_0, \delta_1}{\begin{array}{r}(e_0 \mapsto \mathtt{f}{:}f_0, \mathtt{g}{:}g_0) * \Sigma_0,\ (e_1 \mapsto \mathtt{f}{:}f_1, \mathtt{g}{:}g_1) * \Sigma_1,\ \epsilon \ \rightsquigarrow\ (e \mapsto \mathtt{f}{:}f, \mathtt{g}{:}g) * \Sigma,\ \epsilon',\ \delta_0,\ \delta_1 \\ (\text{when } (e_0, e_1, e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ \mathsf{comb}_\epsilon(f_0, f_1){=}f \ \wedge\ \mathsf{comb}_\epsilon(g_0, g_1){=}g)\end{array}}\ \textsf{match1}$$

$$\frac{\Sigma_0,\ \Sigma_1,\ \mathsf{ext}(\epsilon, f_0, f_1, f) \ \rightsquigarrow\ \Sigma,\ \epsilon',\ \delta_0,\ \delta_1}{\begin{array}{r}\mathsf{ls}\, k_0\, \phi_0\, e_0\, f_0 * \Sigma_0,\ \mathsf{ls}\, k_1\, \phi_1\, e_1\, f_1 * \Sigma_1,\ \epsilon \ \rightsquigarrow\ \mathsf{ls}\,(k_0 \sqcup k_1)\,(\phi_0 \sqcup \phi_1)\, e\, f * \Sigma,\ \epsilon',\ \delta_0,\ \delta_1 \\ (\text{when } (e_0, e_1, e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ \phi_0 \sqcup \phi_1 \text{ is defined} \ \wedge\ \mathsf{comb}_\epsilon(f_0, f_1) = f)\end{array}}\ \textsf{match2}$$

$$\frac{\Sigma_0,\ \Sigma_1,\ \mathsf{ext}(\epsilon, f_0, f_1, f) \ \rightsquigarrow\ \Sigma,\ \epsilon',\ \delta_0,\ \delta_1}{\begin{array}{r}\phi_0(e_0, f_0)[\vec{e}/\vec{x'}] * \Sigma_0,\ (\mathsf{ls}\, k\, \phi_1\, e_1\, f_1) * \Sigma_1,\ \epsilon \ \rightsquigarrow\ (\mathsf{ls}\, k\,(\phi_0 \sqcup \phi_1)\, e\, f) * \Sigma,\ \epsilon',\ \delta_0,\ \delta_1 \\ (\text{when } (e_0, e_1, e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ \phi_0 \sqcup \phi_1 \text{ is defined} \ \wedge\ \mathsf{comb}_\epsilon(f_0, f_1) = f)\end{array}}\ \textsf{match3}$$

$$\frac{\Sigma_0,\ \Sigma_1,\ \mathsf{ext}(\epsilon, f_0, f_1, f) \ \rightsquigarrow\ \Sigma,\ \epsilon',\ \delta_0,\ \delta_1}{\begin{array}{r}(\mathsf{ls}\, k\, \phi_0\, e_0\, f_0) * \Sigma_0,\ \phi_1(e_1, f_1)[\vec{e}/\vec{x'}] * \Sigma_1,\ \epsilon \ \rightsquigarrow\ (\mathsf{ls}\, k\,(\phi_0 \sqcup \phi_1)\, e\, f) * \Sigma,\ \epsilon',\ \delta_0,\ \delta_1 \\ (\text{when } (e_0, e_1, e) \in (\epsilon \cup \mathsf{EQ}) \ \wedge\ \phi_0 \sqcup \phi_1 \text{ is defined} \ \wedge\ \mathsf{comb}_\epsilon(f_0, f_1) = f)\end{array}}\ \textsf{match4}$$

Here $-[\vec{e}/\vec{x'}]$ in $\phi(e, f)[\vec{e}/\vec{x'}]$ is the substitution of all the existentially quantified primed variables $\vec{x'}$ in $\phi(e, f)$ by $\vec{e}$.

Fig. 3. Sample rules for $\rightsquigarrow$. Composite structure case.

program point, which represents the disjunction of those heaps. ⊔ is applied to reduce the number of disjuncts in those sets.

Given a set of symbolic heaps at a program point, the analysis takes two symbolic heaps in the set and applies ⊔ to them. If the application succeeds, the result of the join replaces those heaps. Otherwise, those two symbolic heaps are returned to the set.

In order to maintain precision in the analysis, we restrict the application of ⊔ to only those program points where controlling the number of disjuncts is crucial. They are (a) the beginning of loops, (b) the end of conditional statements when those statements are not inside loops, (c) the call sites of procedures, and (d) the exit points of procedures. The first case accelerates the analysis of the usual fixed-point computation for loops, and the second prevents the combinatorial explosion caused by a sequence of conditional statements; for instance, the procedure `register_cdrom` in `cdrom.c` uses 25 conditional statements to adjust values of a structure for cdrom, which makes the analysis without join suffer from a serious performance problem. The other two cases aim for computing small procedure summaries; the third reduces the number of input symbolic heaps to consider for each procedure, and the last reduces the analysis results of a procedure with respect to each symbolic heap.

We have measured the effects of ⊔ on the performance of SpaceInvader, using our seven driver examples. Table 3 reports the results of our measurements.

Program	LOC	No Opt. (sec)	Opt. except ⧺ (sec)	Opt. including ⧺, with NE only (sec)	Opt. including ⧺, with NE and PE (sec)
scull.c	1010	1.41	1.15	0.59	0.36
class.c	1983	X	X	48.24	8.21
pci-driver.c	2532	X	X	2.69	0.97
ll_rw_blk.c	5469	X	X	X	887.94
cdrom.c	6218	X	X	193.01	103.26
md.c	6653	X	X	X	1585.69
t1394Diag.c	10240	X	X	3415.76	135.05

Table 3. Experimental results on the effects of ⧺. Timeout (X) set at 90min. Experiments run on Intel Core Duo 2.0GHz with 2GB RAM. The "Opt. except ⧺" column records the results of the analysis runs without ⧺ nor possibly empty ls predicates, but with two optimizations: discarding the intermediate analysis results and passing only the reachable portion of the heap to a procedure. The next column contains the analysis time with these two optimizations and ⧺, but without possibly empty ls predicates. The last column contains the analysis time with all the optimizations.

The third and fourth columns of the table record the time of analyzing the drivers without using ⧺: without ⧺, we cannot analyze our example drivers except the simplest one, scull.c. The next two columns concern a pivotal design decision for ⧺, looking at variations on the ls predicate; the fifth column considers the necessarily non-empty ls predicate only, and the sixth column considers both the necessarily non-empty and possibly empty ls predicates. These experimental results confirm the benefit of using the ls PE predicate in ⧺.

5 Related Work

Device driver verification has attracted considerable interest due to the realization that most OS failures arise from bugs in device drivers [8, 24, 2]. Tools like Slam [2] and Blast [15] have been effectively applied in verification of properties of real device drivers, especially properties describing the calling conventions of OS kernel APIs. Unfortunately these tools use coarse models of the heap; Slam, for example, *assumes* memory safety. Other tools are known to prove memory safety, but with the restriction that the input programs do not perform dynamic memory allocation (*e.g.* ASTRÉE [6]). Proving full memory safety (which includes array bounds errors as well as what we have termed pointer safety) of entire systems programs is thus a more difficult problem than that considered in this paper, or in work that concentrates on array bounds errors.

Several papers report on the results of applying shape analysis to the source code of substantial, real-world systems programs. The analysis in [14] has been applied to non-trivial code, but the abstract domain there is purposely much less precise than here, and it could not be used to verify pointer safety of the device drivers that we consider. [7] includes an analysis of a restricted and modified version of the Linux scull driver. Our analysis terminates on the modified scull

code (which they kindly supplied to us) in 0.36sec, where [7] terminated in 9.71sec when using user-supplied assertions (which we did not use) to help the analysis along. It is also worth mentioning [13], which uses slicing to remove heap-irrelevant statements. An earlier version of SpaceInvader [3] analyzed several procedures from the 1394 driver used in Table 1. It timed out on an 1800 LOC subset of the driver, and this drove us to consider ⊎.

The very idea of a join operator is of course not novel, and many other joins have been successfully applied in their application domain. The problem is always one of balancing precision and speed. The claim that ⊎ does not lose too much precision is backed up with experimental results. ⊎ is not unrelated to other join operators that have been proposed in shape analysis [19, 1, 7]. For instance, Chang *et al.* define a partial join operator for separation logic formulas, and Arnold [1] develops a notion of "loose embedding" in TVLA [16] which is in an intuitive sense related to our use of predicates for possibly-empty, rather than only nonempty, lists. However, our ⊎ is different in its detailed formulation; unlike Chang *et al.*, we simplify symbolic heaps before applying ⊎, and unlike Arnold and Manevich [19, 1], our ⊎ keeps the structure of composite data structures precisely. The latter difference, in particular, is crucial to verifying the drivers.

Marron *et al.* reports on shape analyses of several Java programs of up to 3705 LOC [20]. They use an aggressive join operator which *always* merges several abstract states into one. Such a join operator would lead to many false alarms when applied to our device drivers (for example, when dealing with exceptional conditions), and so is too imprecise for our goal of *proving* pointer safety.

6 Conclusions

This paper has presented the first application of shape analysis to a real-world industrial verification problem: proving pointer safety of entire Windows and Linux device drivers. We have achieved this milestone by enhancing our separation domain based shape analysis tool with a sophisticated new join operation, ⊎. This paper has made two contributions: ⊎, and a demonstration that shape analysis can be scaled to real-world industrial verification problems. The second contribution is, in a sense, the most important one. We hope, now that we know that whole device drivers can be accurately handled by today's shape analysis tools, that future research papers on the subject will use device drivers and other substantial systems programs as a part of their experimental evaluations.

Acknowledgments. We would like to thank Viktor Vafeiadis for helpful discussions on the OCaml garbage collector. The London authors acknowledge the support of the EPSRC. Lee was supported by Brain Korea 21. Distefano was supported by a Royal Academy of Engineering research fellowship. O'Hearn was supported by a Royal Society Wolfson Research Merit Award.

References

1. G. Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *SAS*, pages 204–220, 2006.
2. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.
3. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *CAV*, 2007.
4. J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
5. J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, 2008.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.
7. B. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS*, 2007.
8. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, 2001.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
10. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
11. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
12. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, 2007.
13. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
14. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
15. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, 2004.
16. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. SAS 2000.
17. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. In *SPACE*, 2006.
18. R. Manevich, T. Lev-Ami, G. Ramalingam, M. Sagiv, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS*, 2008.
19. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.
20. M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, 2008.
21. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
23. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.
24. M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, 2003.