Purity Analysis: An Abstract Interpretation Formulation

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani

Microsoft Research, India.
t-rakand,grama,kapilv@microsoft.com

Abstract. Salcianu and Rinard present a compositional purity analysis that computes a summary for every procedure describing its side-effects. In this paper, we formalize a generalization of this analysis as an abstract interpretation, present several optimizations and an empirical evaluation showing the value of these optimizations. The Salcianu-Rinard analysis makes use of abstract heap graphs, similar to various heap analyses and computes a shape graph at every program point of an analyzed procedure. The key to our formalization is to view the shape graphs of the analysis as an *abstract state transformer* rather than as a set of abstract states: the concretization of a shape graph is a function that maps a concrete state to a set of concrete states. The abstract interpretation formulation leads to a better understanding of the algorithm. More importantly, it makes it easier to change and extend the basic algorithm, while guaranteeing correctness, as illustrated by our optimizations.

1 Introduction

Compositional or modular analysis [6] is a key technique for scaling static analysis to large programs. Our interest is in techniques that analyze a procedure in isolation, using pre-computed summaries for called procedures, computing a summary for the analyzed procedure. Such analyses are widely used and have been found to scale well. In this paper we consider an analysis presented by Salcianu and Rinard [17], based on a pointer analysis due to Whaley and Rinard [19], which we will refer to the WSR analysis. Though referred to as a purity analysis, it is a more general-purpose analysis that computes a summary for every procedure, in the presence of dynamic memory allocation, describing its side-effects. This is one of the few heap analyses that is capable of treating procedures in a compositional fashion.

WSR analysis is interesting for several reasons. Salcianu and Rinard present an application of the analysis to classify a procedure as *pure* or *impure*, where a procedure is impure if its execution can potentially modify pre-existing state. Increasingly, new language constructs (such as iterators, parallel looping constructs and SQL-like query operators) are realized as higher-order library procedures with procedural parameters that are expected to be side-effect free. Purity checkers can serve as verification/bug-finding tools to check usage of these constructs. Our interest in this analysis stems from our use of an extension of this analysis to statically verify the correctness of the use of speculative parallelism [13]. WSR analysis can also help more sophisticated verification tools, such as [8], which use simpler analyses to identify procedure calls that do not affect properties of interest to the verifier and can be abstracted away.

However, we felt the need for various extensions of the WSR analysis. A key motivation was efficiency. Real-world applications make use of large libraries such as the base class libraries in .NET. While the WSR analysis is reasonably efficient, we find that it still does not scale to such libraries. Another motivation is increased functionality: our checker for speculative parallelism [13] needs some extra information (must-write sets) beyond that computed by the analysis. A final motivating factor is better precision: the WSR analysis declares "pure" procedures that use idioms like lazy initialization and caching as impure.

The desire for these extensions leads us to formulate, in this paper, the WSR analysis as an abstract interpretation, to simplify reasoning about the soundness of these extensions. The formulation of the WSR analysis as an abstract interpretation is, in fact, mentioned as an open problem by Salcianu ([16], page 128).

The WSR analysis makes use of abstract heap graphs, similar to various heap analyses and computes a shape graph g_u at every program point u of an analyzed procedure. The key to our abstract interpretation formulation, however, is to view a shape graph utilized by the analysis as an *abstract state transformer* rather than as a set of abstract states: thus, the concretization of a shape graph is a function that maps a concrete state to a set of concrete states. Specifically, if the graph computed at program point u is g_u, then for any concrete state σ, $\gamma(g_u)(\sigma)$ conservatively approximates the set of states that can arise at program point u in the execution of the procedure on an initial state σ. In our formalization, we present a concrete semantics in the style of the functional approach to interprocedural analysis presented by Sharir and Pnueli. The WSR analysis can then be seen as a natural abstract interpretation of this concrete semantics.

We then present three optimizations viz. duplicate node merging, summary merging, and safe node elimination, that improve the efficiency of WSR analysis. We use the abstract interpretation formulation to show that these optimizations are sound. Our experiments show that these optimizations significantly reduce both analysis time (sometimes by two orders of magnitude or more) and memory consumption, allowing the analysis to scale to large programs.

2 The Language, Concrete Semantics, And The Problem

Syntax A program consists of a set of procedures. A procedure P consists of a control-flow graph, with an entry vertex $entry(P)$ and an exit vertex $exit(P)$. The entry vertex has no predecessor and the exit vertex has no successor. Every edge of the control-flow graph is labelled by a primitive statement. The set of primitive statements are shown in Fig. 1. We use $u \overset{S}{\to} v$ to indicate an edge in the control-flow graph from vertex u to vertex v labelled by statement S.

Concrete Semantics Domain Let *Vars* denote the set of variable names used in the program, partitioned into the following disjoint sets: the set of global

Statement S	**Concrete semantics** $[\![S]\!]_c(\mathsf{V}, \mathsf{E}, \sigma)$
$v_1 = v_2$	$\{(\mathsf{V}, \mathsf{E}, \sigma[v_1 \mapsto \sigma(v_2)]\}$
$v = new\ C$	$\{(\mathsf{V} \cup \{n\}, \mathsf{E} \cup \{n\} \times \mathit{Fields} \times \{\mathit{null}\}, \sigma[v \mapsto n]) \mid n \in N_c \setminus \mathsf{V}\}$
$v_1.f = v_2$	$\{(\mathsf{V}, \{\langle u, l, v\rangle \in \mathsf{E} \mid u \neq \sigma(v_1) \vee l \neq f\} \cup \{\langle \sigma(v_1), f, \sigma(v_2)\rangle\}, \sigma)\}$
$v_1 = v_2.f$	$\{(\mathsf{V}, \mathsf{E}, \sigma[v_1 \mapsto n]) \mid \langle \sigma(v_2), f, n\rangle \in \mathsf{E}\}$
$Call\ P(v_1, \cdots, v_k)$	Semantics defined below

Fig. 1. Primitive statements and their concrete semantics

variables *Globals*, the set of local variables *Locals* (assumed to be the same for every procedure), and the set of formal parameter variables *Params* (assumed to be the same for every procedure). Let *Fields* denote the set of field names used in the program. We use a simple language in which all variables and fields are of pointer type. We use a fairly common representation of the concrete state as a concrete (points-to or shape) graph.

Let N_c be an unbounded set of locations used for dynamically allocated objects. A concrete state or points-to graph $g \in \mathbb{G}_c$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq N_c$ represents the set of objects in the heap, $\mathsf{E} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ (a set of labelled edges) represents values of pointer fields in heap objects, and $\sigma \in \Sigma_c = \mathit{Vars} \mapsto \mathsf{V}$ represents the values of program variables. In particular, $(u, f, v) \in \mathsf{E}$ iff the f field of the object u points to object v. We assume N_c includes a special element *null*. Variables and fields of new objects are initialized to *null*.

Let $\mathcal{F}_c = \mathbb{G}_c \mapsto 2^{\mathbb{G}_c}$ be the set of functions that map a concrete state to a set of concrete states. We define a partial order $\sqsubseteq_c$ on $\mathcal{F}_c$ as follows: $f_a \sqsubseteq_c f_b$ iff $\forall g \in \mathbb{G}_c.f_a(g) \subseteq f_b(g)$. Let $\sqcup_c$ denote the corresponding least upper bound (join) operation defined by: $f_a \sqcup_c f_b = \lambda g.f_a(g) \cup f_b(g)$. For any $f \in \mathcal{F}_c$, we define $\overline{f} : 2^{\mathbb{G}_c} \mapsto 2^{\mathbb{G}_c}$ by: $\overline{f}(G) = \cup_{g \in G} f(g)$. We define the "composition" of two functions in $\mathcal{F}_c$ as follows: $f_a \circ f_b = \lambda g.\overline{f_b}(f_a(g))$.

Concrete Semantics Every primitive statement S has a semantics $[\![S]\!]_c \in \mathcal{F}_c$, as shown in Fig. 1. Every primitive statement has a label ℓ which is not used in the concrete semantics and is, hence, omitted from the figure. The execution of most statements transforms a concrete state to another concrete state, but the signature allows us to model non-determinism (e.g., dynamic memory allocation can return any unallocated object). The signature also allows us to model execution errors such as null-pointer dereference, though the semantics presented simplifies error handling by treating *null* as just a special object.

We now define a concrete summary semantics $[\![P]\!]_c \in \mathcal{F}_c$ for every procedure P. The semantic function $[\![P]\!]_c$ maps every concrete state g_c to the set of concrete states that the execution of P with initial state g_c can produce.

We introduce a new variable φ_u for every vertex in the control-flow graph (of any procedure) and a new variable $\varphi_{u,v}$ for every edge $u \to v$ in the control-flow graph. The semantics is defined as the least fixed point of the following set of equations. The value of φ_u in the least fixed point is a function that maps any concrete state g to the set of concrete states that arise at program point u when the procedure containing u is executed with an initial state g. Similarly, $\varphi_{u,v}$

captures the states after the execution of the statement labelling edge $u \to v$.

$$
\begin{array}{rll}
\varphi_v = \lambda g.\{g\} & v \text{ is an entry vertex} & (1)\\
\varphi_v = \bigsqcup_c\{\varphi_{u,v} \mid u \to v\} & v \text{ is not an entry vertex} & (2)\\
\varphi_{u,v} = \varphi_u \circ [\![S]\!]_c & \text{where } u \xrightarrow{S} v \text{ and S is not a call-stmt} & (3)\\
\varphi_{u,v} = \varphi_u \circ \mathit{CallReturn}_S(\varphi_{exit(Q)}) & \text{where } u \xrightarrow{S} v,\ \text{S is a call to proc Q} & (4)
\end{array}
$$

The first three equations are straightforward. Consider Eq. 4, corresponding to a call to a procedure Q. The value of $\varphi_{exit(Q)}$ summarizes the effect of the execution of the whole procedure Q. In the absence of local variables and parameters, we can define the right-hand-side of the equation to be simply $\varphi_u \circ \varphi_{exit(Q)}$.

The function $\mathit{CallReturn}_S(f)$, defined below, first initializes values of all local variables (to *null*) and formal parameters (to the values of corresponding actual parameters), using an auxiliary function push_S. It then applies f, capturing the procedure call's effect. Finally, the original values of local variables and parameters (of the calling procedure) are restored from the state preceding the call, using a function pop_S. For simplicity, we omit return values from our language.

Let $\mathit{Param}(i)$ denote the i-the formal parameter. Let S be a procedure call statement "`Call Q(`a_1`,...,`a_k`)`". We define the functions $\mathit{push}_S \in \Sigma_c \mapsto \Sigma_c$, $\mathit{pop}_S \in \Sigma_c \times \Sigma_c \mapsto \Sigma_c$, and $\mathit{CallReturn}_S$ as follows:

$$
\begin{aligned}
\mathit{push}_S(\sigma) &= \lambda v.\ v \in \mathit{Globals} \to \sigma(v) \mid v \in \mathit{Locals} \to \mathit{null} \mid v = \mathit{Param}(i) \to \sigma(a_i)\\
\mathit{pop}_S(\sigma, \sigma') &= \lambda v.\ v \in \mathit{Globals} \to \sigma'(v) \mid v \in \mathit{Locals} \cup \mathit{Params} \to \sigma(v)\\
\mathit{CallReturn}_S(f) &= \lambda(\mathsf{V}, \mathsf{E}, \sigma).\{(\mathsf{V}', \mathsf{E}', \mathit{pop}_S(\sigma, \sigma')) \mid (\mathsf{V}', \mathsf{E}', \sigma') \in f(\mathsf{V}, \mathsf{E}, \mathit{push}_S(\sigma))\}
\end{aligned}
$$

We define $[\![P]\!]_c$ to be the value of $\varphi_{exit(P)}$ in the least fixed point of equations (1)-(4), which exists by Tarski's fixed point theorem. Specifically, let VE denote the set of vertices and edges in the given program. The above equations can be expressed as a single equation $\varphi = F^\natural(\varphi)$, where $F^\natural$ is a monotonic function from the complete lattice $VE \mapsto \mathcal{F}_c$ to itself. Hence, $F^\natural$ has a least fixed point.

We note that the above collection of equations is similar to those used in Sharir and Pnueli's functional approach to interprocedural analysis [18] (extended by Knoop and Steffen [10]), with the difference that we are defining a concrete semantics here, while [18] is focused on abstract analyses. The equations are a simple functional version of the standard equations for defining a collecting semantics, with the difference that we are simultaneously computing a collecting semantics for every possible initial states of the procedure's execution.

The goal of the analysis is to compute an approximation of the set of quantities $[\![P]\!]_c$ using abstract interpretation.

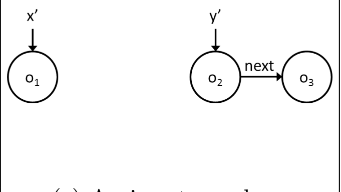
3 The WSR Analysis As An Abstract Interpretation

3.1 Transformer Graphs: An Informal Overview

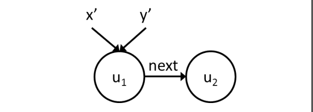
The WSR analysis uses a single abstract graph to represent a set of concrete states, similar to several shape and pointer analyses. The distinguishing aspect of the WSR analysis, however, is its extension of the graph based representation to represent (abstractions of) elements belonging to the functional domain $\mathcal{F}_c$. We now illustrate, using an example, how the graph representation is extended to represent an element of $\mathcal{F}_c = \mathbb{G}_c \mapsto 2^{\mathbb{G}_c}$. Consider the example procedure P shown in Fig. 2(a).

```
    P (x, y) {
 [1]   t = new ();
 [2]   x.next = t;
 [3]   t.next = y;
 [4]   retval = y.next;
    }
```

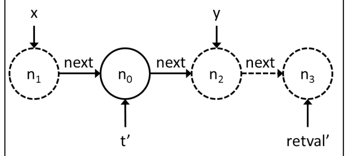
(a) Example procedure P



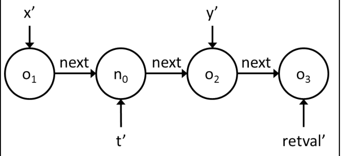
(c) An input graph g_1



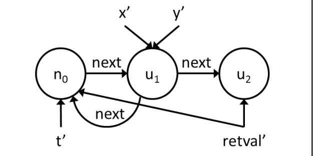
(e) Input graph g_2



(b) Summary graph τ



(d) Output graph $g'_1 = \tau\langle g_1 \rangle$



(f) Output graph $g'_2 = \tau\langle g_2 \rangle$

Fig. 2. Illustration of transformer graphs.

The summary graph τ computed for this procedure is shown in Fig. 2(b). (We omit the *null* node from the figures to keep them simple.) Vertices in a summary graph are of two types: *internal* (shown as circles with a solid outline) and *external* nodes (shown as circles with a dashed outline). Internal nodes represent new heap objects created during the execution of the procedure. E.g., vertex n_0 is an internal node and represents the object allocated in line 1. External nodes, in many cases, represent objects that exist in the heap when the procedure is invoked. In our example, n_1, n_2, and n_3 are external nodes.

Edges in the graph are also classified into *internal* and *external* edges, shown as solid and dashed edges respectively. The edges $n_1 \rightarrow n_0$ and $n_0 \rightarrow n_2$ are internal edges. They represent updates performed by the procedure (i.e., new points-to edges added by the procedure's execution) in lines 2 and 3. Edge $n_2 \rightarrow n_3$ is an external edge created by the dereference "`y.next`" in line 4. This edge helps identify the node(s) that the external node n_3 represents: namely, the objects obtained by dereferencing the `next` field of objects represented by n_2.

The summary graph τ indicates how the execution of procedure P transforms an initial concrete state. Specifically, consider an invocation of procedure P in an initial state given by graph g_1 shown in Fig. 2(c). The summary graph helps construct a transformed graph $g'_1 = \tau\langle g_1 \rangle$, corresponding to the state after the procedure's execution (shown in Fig. 2(d)) by identifying a set of new nodes and edges that must be added to g_1. (The underlying analysis performs no strong updates on the heap and, hence, never removes nodes or edges from the graph). We add a new vertex to g_1 for every internal node n in the summary graph.

Every external node n in the summary graph represents a set of vertices $\eta(n)$ in g_1'. (We will explain later how the function η is determined by τ.) Every internal edge $u \overset{h}{\to} v$ in the summary graph identifies a set of edges $\{u' \overset{h}{\to} v' \mid u' \in \eta(u), v' \in \eta(v)\}$ that must be added to the graph g_1'. In our example, n_1, n_2 and n_3 represent, respectively, $\{o_1\}$, $\{o_2\}$ and $\{o_3\}$. This produces the graph shown in Fig. 2(d), which is an *abstract* graph representing a set of concrete states. The primed variables in the summary graph represent the (final) values of variables, and are used to determine the values of variables in the output graph.

An important aspect of the summary computed by the WSR analysis is that it can be used even in the presence of potential aliases in the input (or cut-points [14]). Consider the input state g_2 shown in Fig. 2(e), in which parameters `x` and `y` point to the same object u_1. Our earlier description of how to construct the output graph still applies in this context. The main tricky aspect here is in correctly dealing with aliasing in the input. In the concrete execution, the update to `x.next` in line 2 updates the `next` field of object u_1. The aliasing between `x` and `y` means that `y.next` will evaluate to n_0 in line 4. Thus, in the concrete execution `retval` will point to the newly created object n_0 at the end of procedure execution, rather than u_2. This complication is dealt with in the definition of the mapping function η. For the example input g_2, the external node n_3 of the summary graph represents the set of nodes $\{u_2, n_0\}$. (This is an imprecise, but sound, treatment of the aliasing situation.) The rest of the construction applies just as before. This yields the abstract graph shown in Fig. 2(f).

More generally, an external node in the summary graph acts as a proxy for a set of *vertices in the final output graph to be constructed*, which may include nodes that exist in the input graph as well as new nodes added to the input graph (which themselves correspond to internal nodes of the summary graph).

We now define the transformer graph domain formally.

3.2 The Abstract Domain

The Abstract Graph Domain We utilize a fairly standard abstract shape (or points-to) graph to represent a set of concrete states. Our formulation is parameterized by a given set N_a, the universal set of all abstract graph nodes. An abstract shape graph $g \in \mathbb{G}_a$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq N_a$ represents the set of abstract heap objects, $\mathsf{E} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ (a set of labelled edges) represents possible values of pointer fields in the abstract heap objects, and $\sigma \in \mathit{Vars} \mapsto 2^{\mathsf{V}}$ is a map representing the possible values of program variables.

Given a concrete graph $g_1 = \langle \mathsf{V}_1, \mathsf{E}_1, \sigma_1 \rangle$ and an abstract graph $g_2 = \langle \mathsf{V}_2, \mathsf{E}_2, \sigma_2 \rangle$ we say that g_1 can be embedded into g_2, denoted $g_1 \preceq g_2$, if there exists a function $h : \mathsf{V}_1 \mapsto \mathsf{V}_2$ such that $\langle x, f, y \rangle \in \mathsf{E}_1 \Rightarrow \langle h(x), f, h(y) \rangle \in \mathsf{E}_2$ and $\forall v \in \mathit{Vars}.\ \sigma_2(v) \supseteq \{h(\sigma_1(v))\}$. The concretization $\gamma_G(g_a)$ of an abstract graph g_a is defined to be the set of all concrete graphs that can be embedded into g_a:

$$\gamma_G(g_a) = \{g_c \in \mathbb{G}_c \mid g_c \preceq g_a\}$$

The Abstract Functional Domain. We now define the domain of graphs used to represent summary functions. A *transformer graph* $\tau \in \mathcal{F}_a$ is a tuple

$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$, where $\mathsf{EV} \subseteq N_a$ is the set of external vertices, $\mathsf{IV} \subseteq N_a$ is the set of internal vertices, $\mathsf{EE} \subseteq V \times \mathit{Fields} \times V$ is the set of external edges, where $V = \mathsf{EV} \cup \mathsf{IV}$, $\mathsf{IE} \subseteq V \times \mathit{Fields} \times V$ is the set of internal edges, $\pi \in (\mathit{Params} \cup \mathit{Globals}) \mapsto 2^V$ is a map representing the values of parameters and global variables in the *initial* state, and $\sigma \in \mathit{Vars} \mapsto 2^V$ is a map representing the possible values of program variables in the *transformed* state. Furthermore, a transformer graph τ is required to satisfy the following constraints:

$$\begin{aligned}\langle x, f, y\rangle \in \mathsf{EE} &\implies \exists u \in \mathit{range}(\pi).\text{x is reachable from u via } (\mathsf{IE} \cup \mathsf{EE}) \text{ edges}\\ y \in \mathsf{EV} &\implies y \in \mathit{range}(\pi) \vee \exists\langle x, f, y\rangle \in \mathsf{EE}\end{aligned}$$

Given a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$, a node u is said to be a parameter node if $u \in \mathit{range}(\pi)$. A node u is said to be an escaping node if it is reachable from some parameter node via a path of zero or more edges (either internal or external). Let $\mathit{Escaping}(\tau)$ denote the set of escaping nodes in τ.

We now define the concretization function $\gamma_T : \mathcal{F}_a \to \mathcal{F}_c$. Given a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$ and a concrete graph $g_c = (\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$, we need to construct a graph representing the transformation of g_c by τ. As explained earlier, every external node $n \in \mathsf{EV}$ in the transformer graph represents a set of vertices in the transformed graph. We now define a function $\eta : (\mathsf{IV} \cup \mathsf{EV}) \mapsto 2^{(\mathsf{IV} \cup \mathsf{V}_c)}$ that maps each node in the transformer graph to a set of concrete nodes (in g_c) as well as internal nodes (in τ) as the least solution to the following set of constraints over variable μ.

$$v \in \mathsf{IV} \Rightarrow v \in \mu(v) \tag{5}$$

$$v \in \pi(\mathtt{X}) \Rightarrow \sigma_c(\mathtt{X}) \in \mu(v) \tag{6}$$

$$\langle u, f, v\rangle \in \mathsf{EE}, u' \in \mu(u), \langle u', f, v'\rangle \in \mathsf{E}_c \Rightarrow v' \in \mu(v) \tag{7}$$

$$\langle u, f, v\rangle \in \mathsf{EE}, \mu(u) \cap \mu(u') \neq \emptyset, \langle u', f, v'\rangle \in \mathsf{IE} \Rightarrow \mu(v') \subseteq \mu(v) \tag{8}$$

Explanation of the constraints: An internal node represents itself (Eq. 5). An external node labelled by a parameter `X` represents the node pointed to by `X` in the input state g_c (Eq. 6). An external edge $\langle u, f, v\rangle$ indicates that v represents any f-successor v' of any node u' represented by u in the input state (Eq. 7). However, with an external edge $\langle u, f, v\rangle$, we must also account for updates to the f field of the objects represented by u during the procedure execution, ie, the transformation represented by τ, via aliases (as illustrated by the example in Fig. 2(e)). Eq. 8 handles this. The precondition identifies u' as a potential alias for u (for the given input graph), and identifies updates performed on the f field of (nodes represented by) u'.

Given mapping function η, we define the transformed abstract graph $\tau\langle g_c\rangle$ as $\langle \mathsf{V}', \mathsf{E}', \sigma'\rangle$, where $\mathsf{V}' = \mathsf{V}_c \cup \mathsf{IV}$, $\mathsf{E}' = \mathsf{E}_c \cup \{\langle v_1, f, v_2\rangle \mid \langle u, f, v\rangle \in \mathsf{IE}, v_1 \in \eta(u), v_2 \in \eta(v)\}$ and $\sigma' = \lambda x. \bigcup_{u \in \sigma(x)} \eta(u)$. The transformed graph is an *abstract* graph that represents all concrete graphs that can be embedded in the abstract graph. Thus, we define the concretization function as below:

$$\gamma_T(\tau_a) = \lambda g_c.\gamma_G(\tau_a\langle g_c\rangle).$$

Our abstract interpretation formulation uses only a concretization function. There is no abstraction function α_T. While this form is less common, it is sufficient to establish the soundness of the analysis, as explained in [5]. Specifically, a concrete value $f \in \mathcal{F}_c$ is *correctly represented* by an abstract value $\tau \in \mathcal{F}_a$, denoted $f \sim \tau$, iff $f \sqsubseteq_c \gamma_T(\tau)$. We seek to compute an abstract value that correctly represents the least fixed point of the concrete semantic equations.

Containment Ordering. A natural "precision ordering" exists on $\mathcal{F}_a$, where τ_1 is said to be more precise than τ_2 iff $\gamma_T(\tau_1) \sqsubseteq_c \gamma_T(\tau_2)$. However, this ordering is not of immediate interest to us. (It is not even a partial order, and is hard to work with computationally.) We utilize a stricter ordering in our abstract fixed point computation. We define a relation $\sqsubseteq_{co}$ on $\mathcal{F}_a$ by: $(\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1) \sqsubseteq_{co} (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2)$ iff $\mathsf{EV}_1 \subseteq \mathsf{EV}_2$, $\mathsf{EE}_1 \subseteq \mathsf{EE}_2$, $\forall x.\pi_1(x) \subseteq \pi_2(x)$, $\mathsf{IV}_1 \subseteq \mathsf{IV}_2$, $\mathsf{IE}_1 \subseteq \mathsf{IE}_2$, and $\forall x.\sigma_1(x) \subseteq \sigma_2(x)$.

Lemma 1. *$\sqsubseteq_{co}$ is a partial-order on $\mathcal{F}_a$ with a join operation, denoted $\sqcup_{co}$. Further, γ_T is monotonic with respect to $\sqsubseteq_{co}$: $\tau_1 \sqsubseteq_{co} \tau_2 \Rightarrow \gamma_T(\tau_1) \sqsubseteq_c \gamma_T(\tau_2)$.*

3.3 The Abstract Semantics

Our goal is to approximate the least fixed point computation of the concrete semantics equations 1-4. We do this by utilizing an analogous set of abstract semantics equations shown below. First, we fix the set N_a of abstract nodes. Recall that the domain $\mathcal{F}_a$ defined earlier is parameterized by this set. The WSR algorithm relies on an "allocation site" based merging strategy for bounding the size of the transformer graphs. We utilize the labels attached to statements as allocation-site identifiers. Let *Labels* denote the set of statement labels in the given program. We define N_a to be $\{n_x \mid x \in \mathit{Labels} \cup \mathit{Params} \cup \mathit{Globals}\}$.

We first introduce a variable ϑ_u for every vertex u in the control-flow graph (denoting the abstract value at a program point u), and a variable $\vartheta_{u,v}$ for every edge $u \to v$ in the control-flow graph (denoting the abstract value after the execution of the statement in edge $u \to v$).

$$\vartheta_v = \mathsf{ID} \qquad v \text{ is an entry vertex} \tag{9}$$

$$\vartheta_v = \sqcup_{co}\{\vartheta_{u,v} \mid u \xrightarrow{S} v\} \qquad v \text{ is not an entry vertex} \tag{10}$$

$$\vartheta_{u,v} = [\![S]\!]_a(\vartheta_u) \qquad \text{where } u \xrightarrow{S} v, \text{S is not a call-stmt} \tag{11}$$

$$\vartheta_{u,v} = \vartheta_{exit(Q)}\langle\!\langle \vartheta_u \rangle\!\rangle_a^S \qquad \text{where } u \xrightarrow{S} v, \text{S is a call to Q} \tag{12}$$

Here, ID is a transformer graph consisting of a external vertex for each global variable and each parameter (representing the identity function). Formally, $\mathsf{ID} = (\mathsf{EV}, \emptyset, \pi, \emptyset, \emptyset, \pi)$, where $\mathsf{EV} = \{n_x \mid x \in \mathit{Params} \cup \mathit{Globals}\}$ and $\pi = \lambda v.\ v \in \mathit{Params} \cup \mathit{Globals} \to n_v \mid v \in \mathit{Locals} \to \mathit{null}$. The abstract semantics $[\![S]\!]_a$ of any primitive statement S, other than a procedure call, is shown in Figure 3. The abstract semantics of a procedure call is captured by an operator $\tau_1\langle\!\langle \tau_2 \rangle\!\rangle_a^S$, which we will define soon.

Statement S	**Abstract semantics** $[\![S]\!]_a\tau$ where $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$
$v_1 = v_2$	$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto \sigma(v_2)])$
$\ell : v = new\ C$	$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV} \cup \{n_\ell\}, \mathsf{IE} \cup \{n_\ell\} \times Fields \times \{null\}, \sigma[v \mapsto \{n_\ell\}])$
$v_1.f = v_2$	$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE} \cup \sigma(v_1) \times \{f\} \times \sigma(v_2), \sigma)$
$\ell : v_1 = v_2.f$	$let\ A = \{n \mid \exists n_1 \in \sigma(v_2), \langle n_1, f, n\rangle \in \mathsf{IE}\}\ in$ $let\ B = \sigma(v_2) \cap Escaping(\tau)\ in$ $if\ (B = \emptyset)$ $then\ (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto A])$ $else\ (\mathsf{EV} \cup \{n_\ell\}, \mathsf{EE} \cup B \times \{f\} \times \{n_\ell\}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v \mapsto A \cup \{n_\ell\}])$

Fig. 3. Abstract semantics of primitive instructions.

The abstract semantics of the first three statements are straightforward. The treatment of the dereference $v_2.f$ in the last statement is more involved. Here, the simpler case is where the dereferenced object is a non-escaping object: in this case, we can directly determine the possible values of $v_2.f$ from the information computed by the local analysis of the procedure. This is handled by the true branch of the conditional statement. The case of escaping objects is handled by the false branch. In this case, in addition to the possible values of $v_2.f$ identified by the local analysis, we must account for two sources of values unknown to the local analysis. The first possibility is that the dereferenced object is a pre-existing object (in the input state) with a pre-existing value for the f field. The second possibility is that the dereferenced object may have aliases unknown to the local analysis via which its f field may have been updated during the procedure's execution. We create an appropriate external node (with a corresponding incoming external edge) that serves as a proxy for these unknown values.

We now consider the abstract semantics of a procedure call statement. Let $\tau_r = (\mathsf{EV}_r, \mathsf{EE}_r, \pi_r, \mathsf{IV}_r, \mathsf{IE}_r, \sigma_r)$ be the transformer graph in the caller before a call statement S to Q and let $\tau_e = (\mathsf{EV}_e, \mathsf{EE}_e, \pi_e, \mathsf{IV}_e, \mathsf{IE}_e, \sigma_e)$ be the abstract summary of Q. We now show how to construct the graph $\tau_e\langle\!\langle\tau_r\rangle\!\rangle^S_a$ representing the abstract graph at the point after the method call. This operation is an extension of the operation $\tau\langle g_c\rangle$ used earlier to show how τ transforms a concrete state g_c into one of several concrete states.

We first utilize an auxiliary transformer $\tau_e\langle\!\langle\tau_r, \eta\rangle\!\rangle$ that takes an extra parameter η that maps nodes of τ_e to a set of nodes in τ_e and τ_r. (As explained above, a node u in τ_e acts as a proxy for a set of vertices in a particular callsite and $\eta(u)$ identifies this set.) Given η, define $\hat{\eta}$ as $\lambda X. \bigcup_{u \in X} \eta(u)$. We then define $\tau_e\langle\!\langle\tau_r, \eta\rangle\!\rangle$ to be $(\mathsf{EV}', \mathsf{EE}', \pi', \mathsf{IV}', \mathsf{IE}', \sigma')$ where

$$\begin{aligned}
\mathsf{V}' &= (\mathsf{IV}_r \cup \mathsf{EV}_r) \cup \hat{\eta}(\mathsf{IV}_e \cup \mathsf{EV}_e) \\
\mathsf{IV}' &= \mathsf{V}' \cap (\mathsf{IV}_r \cup \mathsf{IV}_e) \\
\mathsf{EV}' &= \mathsf{V}' \cap (\mathsf{EV}_r \cup \mathsf{EV}_e) \\
\mathsf{IE}' &= \mathsf{IE}_r \cup \{\langle v_1, f, v_2\rangle \mid \langle u, f, v\rangle \in \mathsf{IE}_e, v_1 \in \eta(u), v_2 \in \eta(v)\} \\
\mathsf{EE}' &= \mathsf{EE}_r \cup \{\langle u', f, v\rangle \mid \langle u, f, v\rangle \in \mathsf{EE}_e, u' \in \eta(u), escapes(u')\} \\
\pi' &= \pi_r
\end{aligned}$$

$$\sigma' = \lambda x.\ x \in \mathit{Globals} \to \hat{\eta}(\sigma_e(x)) \mid x \in \mathit{Locals} \cup \mathit{Params} \to \sigma_r(x)$$
$$\mathit{escapes}(v) \equiv \exists u \in \mathit{range}(\pi').v \text{ is reachable from } u \text{ via } \mathsf{IE}' \cup \mathsf{EE}' \text{ edges}$$

The predicate "escapes(u')" used in the above definition is recursively dependent on the graph τ' being constructed: it checks if u' is reachable from any of the parameter nodes in the graph being constructed. Thus, this leads to an iterative process for adding edges to the graph being constructed, as more escaping nodes are identified.

We now show how the node mapping function η is determined, given the transformers τ_e and τ_r. The function η is defined to be the least fixed point of the set of following constraints over the variable μ. (Here, μ_1 is said to be less than μ_2 iff $\mu_1(u) \subseteq \mu_2(u)$ for all u.) Let a_i denote the actual argument corresponding to the formal argument $\mathit{Param}(i)$.

$$x \in \mathsf{IV}_e \Rightarrow x \in \mu(x) \tag{13}$$
$$x \in \pi_e(\mathit{Param}(i)) \Rightarrow \sigma_r(a_i) \subseteq \mu(x) \tag{14}$$
$$x \in \pi_e(v) \wedge v \in \mathit{Globals} \Rightarrow \sigma_r(v) \subseteq \mu(x) \tag{15}$$
$$\langle u, f, v\rangle \in \mathsf{EE}_e, u' \in \mu(u), \langle u', f, v'\rangle \in \mathsf{IE}_r \Rightarrow v' \in \mu(v) \tag{16}$$
$$\langle u, f, v\rangle \in \mathsf{EE}_e, \mu(u) \cap \mu(u') \neq \emptyset, \langle u', f, v'\rangle \in \mathsf{IE}_e \Rightarrow \mu(v') \subseteq \mu(v) \tag{17}$$
$$\langle u, f, v\rangle \in \mathsf{EE}_e, \mu(u) \cap \mathit{Escaping}(\tau_e\langle\!\langle \tau_r, \mu\rangle\!\rangle) \neq \emptyset \Rightarrow v \in \mu(v) \tag{18}$$

In WSR analysis, rule (17) has one more pre-condition, namely ($u \neq u' \vee u \in \mathsf{EV}_e$). This extra condition may result in a more precise node mapping function but requires a similar change to the definition of the concretization function γ_T.

Abstract Fixed Point Computation. The collection of equations 9-12 can be viewed as a single equation $\vartheta = F^\sharp(\vartheta)$, where $F^\sharp$ is a function from $VE \mapsto \mathcal{F}_a$ to itself. Let $\perp$ denote $\lambda x.(\{\}, \{\}, \lambda v.\{\}, \{\}, \{\}, \lambda v.\{\})$. The analysis iteratively computes the sequence of values ${F^\sharp}^i(\perp)$ and terminates when ${F^\sharp}^i(\perp)$ $= {F^\sharp}^{i+1}(\perp)$. We define $[\![P]\!]_a$ (the summary for a procedure P) to be the value of $\varphi_{exit(P)}$ in the final solution.

Correctness and Termination. With this formulation, correctness and termination of the analysis follow in the standard way. Correctness follows by establishing that $F^\sharp$ is a sound approximation of $F^\natural$, which follows from the following lemma that the corresponding components of $F^\sharp$ are sound approximations of the corresponding components of $F^\natural$. As usual, we say that a concrete value $f \in \mathcal{F}_c$ is *correctly represented* by an abstract value $\tau \in \mathcal{F}_a$, denoted $f \sim \tau$, iff $f \sqsubseteq_c \gamma_T(\tau)$.

Lemma 2. *(a)* $\lambda g.\{g\} \sim \mathsf{ID}$
(b) For every primitive statement S (other than a procedure call), $[\![S]\!]_a$ is a sound approximation of $[\![S]\!]_c$: if $f \sim \tau$, then $f \circ [\![S]\!]_c \sim [\![S]\!]_a(\tau)$.
(c) $\sqcup_{co}$ is a sound approximation of $\sqcup_c$: if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $(f_1 \sqcup_c f_2) \sim (\tau_1 \sqcup_{co} \tau_2)$.
(d) if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $f_2 \circ \mathit{CallReturn}_S(f_1) \sim \tau_1\langle\!\langle \tau_2\rangle\!\rangle_a^S$.

Lemma 2 implies the following soundness theorem in the standard way (e.g., see Proposition 4.3 of [5]).

Theorem 1. *The computed procedure summaries are correct. (For every procedure P, $[\![P]\!]_c \sim [\![P]\!]_a$.)*

Termination follows by establishing that $F^\sharp$ is monotonic with respect to $\sqsubseteq_{co}^*$, since $\mathcal{F}_a$ has only finite height $\sqsubseteq_{co}$-chains. Proofs of all results appear in [11].

4 Optimizations

We have implemented the WSR analysis for .NET binaries. More details about the implementation and how we deal with language features absent in the core language used in our formalization appear in [11]. In this section we describe three optimizations for the analysis that were motivated by our implementation experience. We do not describe optimizations already discussed by WSR in [19] and [17]. We present an empirical evaluation of the impact of these optimizations on the scalability and the precision of the purity analysis in the experimental evaluation section.

Optimization 1: Node Merging. Informally, we define node merging as an operation that replaces a set of nodes $\{n_1, n_2 \dots n_m\}$ by a single node n_{rep} such that any predecessor or successor of the nodes $n_1, n_2, \dots, n_m$ becomes, respectively, a predecessor or successor of n_{rep}. While merging nodes seems like a natural heuristic for improving efficiency, it does introduce some subtle issues and challenges. The intuition for merging nodes arises from their use in the context of heap analyses where graphs represent sets of concrete states. However, in our context, graphs represent state transformers. We now present some results that help establish the correctness of this optimization.

We now extend the notion of graph embedding to transformer graphs. Given $\tau_1 = (\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1)$ and $\tau_2 = (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2)$, we say that $\tau_1 \preceq \tau_2$ iff there exists a function $h : (\mathsf{IV}_1 \cup \mathsf{EV}_1) \mapsto (\mathsf{IV}_2 \cup \mathsf{EV}_2)$ such that: for every internal (respectively, external) node x in τ_1 , $h(x)$ is an internal (respectively, external) node; for every internal (respectively, external) edge $\langle x, f, y\rangle$ in τ_1, $\langle h(x), f, h(y)\rangle$ is an internal (respectively, external) edge in τ_2, for every variable $\mathtt{x}$, $\hat{h}(\sigma_1(\mathtt{x})) \subseteq \sigma_2(\mathtt{x})$ and $\hat{h}(\pi_1(\mathtt{x})) \subseteq \pi_2(\mathtt{x})$ where $\hat{h}(Z) = \{h(u) \mid u \in Z\}$.

Node merging produces an embedding. Assume that we are given an equivalence relation $\simeq$ on the nodes of a transformer graph τ (such that no internal nodes are equivalent to external nodes). We define the transformer graph $\tau/\simeq$ to be the transformer graph obtained by replacing every node u by a unique representative of its $\simeq$-equivalence class in every component of τ.

Lemma 3. *(a) $\preceq$ is a pre-order. (b) γ_T is monotonic with respect to $\preceq$: i.e., $\forall \tau_a, \tau_b \in \mathcal{F}_a . \tau_a \preceq \tau_b \Rightarrow \gamma_T(\tau_a) \sqsubseteq_c \gamma_T(\tau_b)$. (c) $\tau \preceq (\tau/\simeq)$.*

Assume that we wish to replace a transformer graph τ by a graph $\tau/\simeq$ at some point during the analysis (perhaps by incorporating this into one of the abstract operations). Our earlier correctness argument still remains valid (since if $f \sim \tau_1 \preceq \tau_2$, then $f \sim \tau_2$).

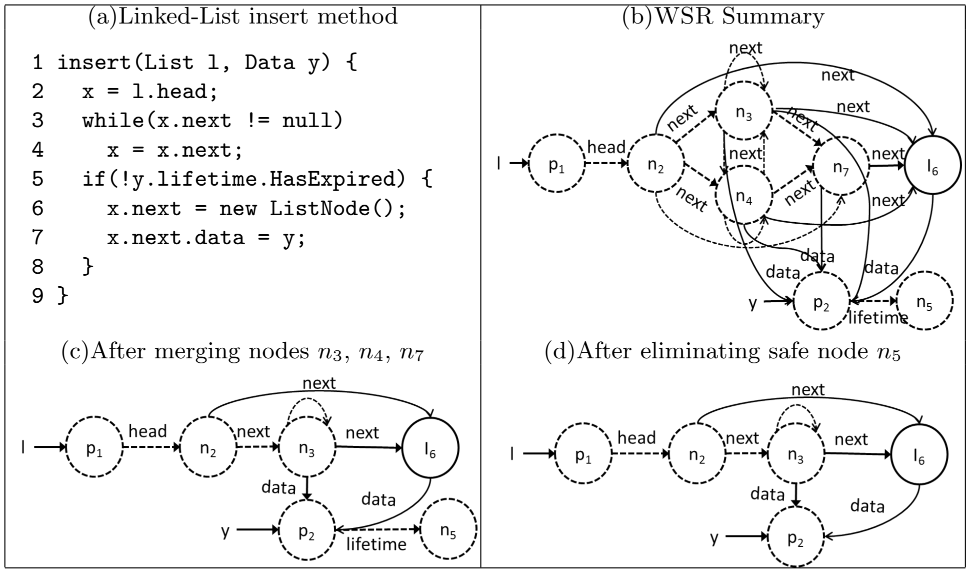
However, this optimization impacts the termination argument because we do not have $\tau \sqsubseteq_{co} (\tau/\simeq)$. Indeed, our initial implementation of the optimization did not terminate for one program because the computation ended up with a cycle of equivalent, but different, transformers (in the sense of having the same concretization). Refining the implementation to ensure that once two nodes are chosen to be merged together, they are always merged together in all subsequent steps, guarantees termination. Technically, we enhance the domain to include an equivalence relation on nodes (representing the nodes currently merged together) and update the transformers accordingly. A suitably modified ordering relation ensures termination. Details are omitted due to space constraints, but this illustrated to us the value of the abstract interpretation formalism (see [11] for more details).

The main advantage of the node merging optimization is that it reduces the size of the transformer graph while every other transfer function increases the size of the transformer graphs. However, when used injudiciously, node merging can result in loss of precision. In our implementation we use a couple of heuristics to identify the set of nodes to be merged.

Given $\tau \in \mathcal{F}_a$ and $v_1, v_2 \in \mathsf{V}(\tau)$, we merge v_1, v_2 iff one of the two conditions hold (a) $v_1, v_2 \in \mathsf{EV}(\tau)$ and $\exists u \in \mathsf{V}(\tau)$ s.t. $\langle u, f, v_1 \rangle \in \mathsf{EE}(\tau)$ and $\langle u, f, v_2 \rangle \in \mathsf{EE}(\tau)$ for some field f or (b) $v_1, v_2 \in \mathsf{IV}(\tau)$ and $\exists u \in \mathsf{V}(\tau)$ s.t. $\langle u, f, v_1 \rangle \in \mathsf{IE}(\tau)$ and $\langle u, f, v_2 \rangle \in \mathsf{IE}(\tau)$ for some field f.

In the WSR analysis, an external edge $\langle u, f, v \rangle$ on an escaping node u is often used to identify objects that $u.f$ may point-to in the state before the call to the method (i.e, pre-state). However, having two external edges with the same source and same field serves no additional purpose. Our first heuristic eliminates such duplicate external edges, which may be produced, e.g., by multiple reads "`x.f`", where `x` is a formal parameter, of the same field of a pre-state object inside a method or its transitive callees. Our second heuristic addresses a similar problem that might arise due to multiple writes to the same field of an internal object inside a method or its transitive callees. Although, theoretically, the above two heuristics can result in loss of precision, it was not the case on most of the programs on which we ran our analysis (see experimental results section). We apply this node-merging optimization only at procedure exit (to the summary graph produced for the procedure).

Figure 4 shows an illustration of this optimization. Figure 4(a) shows a simple procedure that appends an element to a linked list. Figure 4(b) shows the WSR summary graph that would result by the straight forward application of the transfer functions presented in the paper. Figure 4(c) shows the impact of applying the node-merging optimization on the WSR summary shown in Figure 4(b). In the WSR summary, it can be seen that the external node n_2 has three outgoing external edges on the field *next* that end at nodes n_3, n_4 and n_7. This is due to the reads of the field *next* in the line numbers 3, 4 and 7. As shown in Figure 4(b) the blow-up due to these redundant edges is substantial


(a)Linked-List insert method
(b)WSR Summary
(c)After merging nodes n_3, n_4, n_7
(d)After eliminating safe node n_5
l
y
p1
p2
n2
n3
n4
n5
n7
l6
head
next
data
lifetime

```
insert(List l, Data y) {
  x = l.head;
  while(x.next != null)
    x = x.next;
  if(!y.lifetime.HasExpired) {
    x.next = new ListNode();
    x.next.data = y;
  }
}
```

Fig. 4. Illustrative example for the optimizations

(even in this small example). Figure 4(c) shows the transformer graph that results after merging the nodes n_3, n_4 and n_7 that are identified as equivalent by our heuristics. Let the transformer graphs shown in Figure 4(b) and Figure 4(c) be τ_a and τ_b respectively. It can be verified that $\gamma(\tau_a) = \gamma(\tau_b)$.

Optimization 2: Summary Merging. Though the analysis described earlier does not consider virtual method calls, our implementation does handle them (explained in [11]). Briefly, a virtual method call is modelled as a conditional call to one of the various possible implementation methods. Let the transformer graph before and after the virtual method call statement be τ_{in} and τ_{out} respectively. Let the summaries of the possible targets of the call be $\tau_1, \tau_2, \ldots \tau_n$. In the unoptimized approach, $\tau_{out} = \tau_1\langle\!\langle\tau_{in}\rangle\!\rangle \sqcup_{co} \ldots \sqcup_{co} \tau_n\langle\!\langle\tau_{in}\rangle\!\rangle$. This optimization constructs a single summary that over-approximates all the callee summaries, as $\tau_{merge} = \tau_1 \sqcup_{co} \ldots \sqcup_{co} \tau_n$ and computes τ_{out} as $\tau_{merge}\langle\!\langle\tau_{in}\rangle\!\rangle$. Since each $\tau_i \preceq \tau_{merge}$ (in fact, $\tau_i \sqsubseteq_{co} \tau_{merge}$), τ_{merge} is a safe over-approximation of the summaries of all callees. Once the graph τ_{merge} is constructed it is cached and reused when the virtual method call instruction is re-encountered during the fix-point computation (provided the targets of the virtual method call do not change across iterations and their summaries do not change). We further apply node merging to τ_{merge} to obtain τ_{mo} which is used instead of τ_{merge}.

Optimization 3: Safe Node Elimination. This optimization identifies certain external nodes that can be discarded from a method's summary without affecting correctness. As motivation, consider a method *Set::Contains*. This method does not mutate the caller's state, but its summary includes several external nodes that capture the "reads" of the method. These extraneous nodes

Benchmark	**LOC**	**Description**
DocX (*dx*)	10K	library for manipulating Word 2007 files
Facebook APIs (*fb*)	21K	library for integrating with Facebook.
Dynamic data display (*ddd*)	25K	real-time data visualization tool
SharpMap (*sm*)	26K	Geospatial application framework
Quickgraph (*qg*)	34K	Graph Data structures and Algorithms
PDfsharp (*pdf*)	96K	library for processing PDF documents
DotSpatial (*ds*)	220K	libraries for manipulating Geospatial data
mscorlib (*ms*)	Unknown	Core C# library
System (*sys*)	Unknown	Core C# library

Fig. 5. benchmark programs

make subsequent operations more expensive. Let m be a method with a summary τ. An external vertex ev is safe in τ iff it satisfies the following conditions for every vertex v transitively reachable from ev: (a) v is not modified by the procedure, and (b) No internal edge in τ ends at v and there exists no variable t such that $v \in \sigma(t)$. (We track modifications of nodes with an extra boolean attached to nodes.) Let $removeSafeNodes(\tau)$ denote transformer obtained by deleting all safe nodes in τ. We can show that $\gamma_T(removeSafeNodes(\tau)) = \gamma_T(\tau)$. Like node merging we perform this optimization only at method exits. Figure 4(d) shows the transformer graph that would result after eliminating safe nodes from the transformer graph shown in Figure 4(c).

5 Empirical Evaluation

We implemented the purity analysis along with the optimizations using *Phoenix* analysis framework for .NET binaries [12]. In our implementation, summary computation is performed using an intra-procedural *flow-insensitive* analysis using the transfer functions described in Figure 3. We chose a flow-insensitive analysis due to the prohibitively large memory requirements of a flow-sensitive analysis when run on large libraries. We believe that the optimizations that we propose will have a bigger impact on the scalability of a flow-sensitive analysis.

Fig. 5 shows the benchmarks used in our evaluation. All benchmarks (except *mscorlib.dll* and *System.dll*) are open source C# libraries[4]. We carried out our experiments on a 2.83 GHz, 4 core, 64 bit Intel Xeon CPU running Windows Server 2008 with 16GB RAM.

We ran our implementation on all benchmarks in six different configurations (except *QuickGraph* which was run on three configurations only) to evaluate our optimizations: (a) base WSR analysis without any optimizations (*base*) (b) base analysis with summary merging (*base+sm*) (c) base analysis with node merging (*base+nm*) (d) base analysis with summary and node merging (*base+nsm*) (e) base analysis with safe node elimination (*base+sf*) (f) base analysis with all optimizations (*base+all*). We impose a time limit of 3 hours for the analysis of each program (except *QuickGraph* where we used a time limit of 8 hours).

Benchmarks	*dx*	*fb*	*ddd*	*pdf*	*sm*	*ds*	*ms*	*sys*	*qg*
# of methods	612	4112	2266	3883	1466	10810	2963	698	3380
Pure methods	340	1924	1370	1515	934	5699	1979	411	2152
time(s)									
base	21	52	4696	5088	∞	∞	108	17	∞
base+sf	19	46	3972	2914	∞	∞	56	16	–
base+sm	6	14	3244	4637	7009	∞	54	5	∞
base+nm	20	46	58	125	615	963	21	16	–
base+nsm	5	9	26	79	181	251	13	4	–
base+all	5	8	23	76	179	232	12	4	21718
memory(MB)									
base	313	478	1937	1502	∞	∞	608	387	∞
base+sf	313	460	1836	1136	∞	∞	545	390	–
base+sm	313	478	1937	1508	369	∞	589	390	∞
base+nm	296	460	427	535	356	568	515	387	–
base+nsm	296	461	411	569	369	568	514	390	–
base+all	296	446	410	550	356	568	497	390	703

Fig. 6. Results of analysing the benchmarks in six configurations

Fig. 6 shows the execution time and memory consumption of our implementation. Runs that exceed the time limit were terminated and their times are listed as ∞. The number of methods classified as pure were same for all configurations (that terminated) for all benchmarks.

The results show that for several benchmarks, node merging drastically reduces analysis time. The other optimizations also reduce the analysis time, though not as dramatically as node merging. Fig. 7 provides insights into the reasons for this improvement by illustrating the correlation between analysis time and number of duplicate edges in the summary. A point (x, y) in the graph indicates that y percentage of analysis time was spent on procedures whose summaries had, on average, at least x outgoing edges per vertex that are labelled by the same field. The benchmarks that benefited from the node merging optimization (viz. SharpMap, PDFSharp, Dynamic Data Display, DotSpatial) spend a large fraction of the analysis time (approx. 90% of the time) on summaries that have average number of duplicate edges per vertex above 4. The graph on the right hand side plots the same metrics when node merging is enabled. It can be seen that node merging is quite effective in reducing the duplicate edges and hence also reduces analysis time.

6 Related Work

Modular Pointer Analyses. The Whaley-Rinard analysis [19], which is the core of Salcianu-Rinard's purity analysis [17], is one of several modular pointer analyses that have been proposed, such as [2] and [3]. Modular pointer analyses offer the promise of scalability to large applications, but are quite complex to understand

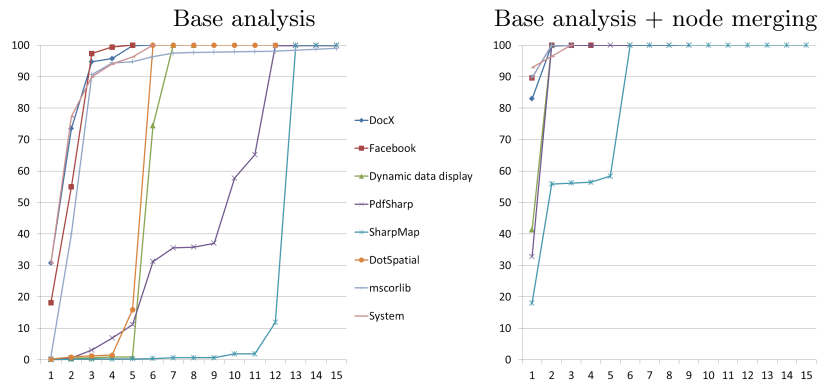


Fig. 7. Number duplicate edges in the summary graph Vs percentage time taken to compute the summary

and implement. We believe that an abstract interpretation formulation of such modular analyses are valuable as they make them accessible to a larger audience and simplify reasoning about variations and modifications of the algorithm. We are not aware of any previous abstract interpretation formulation of a modular pointer analysis. Our formulation also connects the WSR approach to Sharir-Pnueli's functional approach to interprocedural analysis [18].

Compositional Shape Analyses. Calcagno *et al.* [1] and Gulavani *et al.* [7] present separation-logic based compositional approaches to shape analysis. They perform more precise analysis but compute Hoare triples, which correspond to conditional summaries: summaries which are valid only in states that satisfy the precondition of the Hoare triple. These summaries typically incorporate significant "non-aliasing" conditions in the precondition. Modular pointer analyses such as WSR have somewhat different goals. They are less precise, but more scalable and produce summaries that can be used in any input state.

Parametric Shape Analyses. TVLA [15] is a parametric abstract interpretation that has been used to formalize a number of heap and shape analyses. The WSR analysis and our formalization seem closely related to the relational approach to interprocedural shape analysis presented by Jeannet *et al.* [9]. The Jeannet *et al.*approach shows how the abstract shape graphs of TVLA can be used to represent abstract graph transformers (using a double vocabulary), which is used for modular interprocedural analysis. Rinetzky *et al.* [14] present a tabulation-based approach to interprocedural heap analysis of cutpoint-free programs (which imposes certain restrictions on aliasing). (While the WSR analysis computes a procedure summary that can be reused at any callsite, the tabulation approach may analyze a procedure multiple times, but reuses analysis results at different callsites if the "input heap" is the same.) However, there are interesting similarities and connections between the WSR approach and the Rinetzky *et al.* approach to merging "graphs" from the callee and the caller.

Modularity In Interprocedural Analysis. While the WSR analysis is modular in the absence of recursion, recursive procedures must be analyzed together. Our experience has shown that large strongly connected components of procedures in the call-graph can be a bottleneck in analyzing large libraries. An interesting direction for future work is to explore techniques that can be used to achieve modularity even in the presence of recursion, e.g., see [6].

References

1. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. pp. 289–300 (2009)
2. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL. pp. 133–146 (1999)
3. Cheng, B.C., Hwu, W.M.W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In: PLDI. pp. 57–69 (2000)
4. Codeplex. http://www.codeplex.com (March 2011)
5. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. 2(4), 511–547 (1992)
6. Cousot, P., Cousot, R.: Modular static program analysis. In: CC. pp. 159–178 (2002)
7. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis. In: SAS. pp. 188–204 (2009)
8. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: FSE. pp. 117–127 (2006)
9. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. ACM Trans. Program. Lang. Syst. 32, 5:1–5:52 (February 2010), http://doi.acm.org/10.1145/1667048.1667050
10. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: CC. pp. 125–140 (1992)
11. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: An abstract interpretation formulation, (Forthcoming) Tech. rep., Microsoft Research, India.
12. Phoenix. https://connect.microsoft.com/Phoenix (March 2011)
13. Prabhu, P., Ramalingam, G., Vaswani, K.: Safe programmable speculative parallelism. In: PLDI. pp. 50–61 (2010)
14. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: SAS. pp. 284–302 (2005)
15. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL. pp. 105–118 (1999)
16. Salcianu, A.D.: Pointer Analysis and its Applications for Java Programs. Master's thesis, Massachusetts institute of technology (2001)
17. Salcianu, A.D., Rinard, M.C.: Purity and side effect analysis for java programs. In: In VMCAI. Springer-Verlag (2005)
18. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. pp. 189–234 (1981)
19. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for java programs. In: OOPSLA. pp. 187–206 (1999)