Precise Interprocedural Side-Effect Analysis

Manuel Geffken, Hannes Saffrich, and Peter Thiemann

Universität Freiburg, Germany
{geffken,saffriha,thiemann}@informatik.uni-freiburg.de

Abstract. A side-effect analysis computes for each program phrase a set of memory locations that may be read or written to when executing this phrase. Our analysis expresses abstract objects, points-to and aliasing information, escape information, and side effects all in terms of a single novel abstract domain, generalized access graphs. This abstract domain represents sets of access paths precisely and compactly. It is suitable for intraprocedural analysis as well as for constructing method summaries for interprocedural analysis.

We implement the side-effect analysis for Java on top of the SOOT framework and report on its application to selected examples.

1 Introduction

A side-effect analysis computes for each program phrase a set of memory locations that may be read or written to when executing this phrase. The results of such an analysis have many uses in practice including the identification of *pure* methods (that have no side effects), of read-only parameters, and of objects that escape from a method. A compiler may perform aggressive code motion on a call to a pure method. Such a method may also be used in a specification [2]. In a concurrent program, methods with disjoint side effects may run in parallel without interfering [1]. Several program analyses require information on side effects of method calls to correctly transfer local analysis results across call sites [7,9,16]. Furthermore, the search space of a software model checker may be reduced by ignoring interleavings of methods with disjoint side effects [5].

Our analysis expresses abstract objects, points-to and aliasing information, escape information, and side effects all in terms of *generalized access graphs*, a novel abstract domain inspired by Deutsch's *symbolic access paths* (SAPs) [8] and Khedker and coworkers' access graphs [15]. A value in this domain represents information about heap-allocated objects using a regular language of access paths in the pre-state of a method. This condensation of information in one domain facilitates an elegant and economic description of the analysis that completely fits into this paper and it simplifies its implementation.

The analysis computes context insensitive may-information for method summaries. From the method summary, it is straightforward to determine whether parameters are read-only, whether a method is pure, and whether any heap-allocated objects may escape from the method.

G. Ciobanu and D. Méry (Eds.): ICTAC 2014, LNCS 8687, pp. 188–205, 2014.

The intraprocedural analysis that computes the method summaries is flow-sensitive and performs strong updates for local and global variables. The interprocedural analysis is an instance of a bottom-up analysis.

We implement the side-effect analysis for Java on top of the SOOT [23] framework and report on preliminary results of its application to selected examples.

1.1 Contributions

– We define the abstract domain of generalized access graphs (GAGs) as an extension of Khedker and coworkers' access graphs [15]. In comparison, our analysis requires fewer and simpler operations on the domain and can make do with one domain to maintain all kinds of information.
– We specify the intraprocedural GAG-based analysis for a CFG representation of a method in an object-based language (without pointer arithmetic). It is integrated with a context-insensitive bottom-up interprocedural analysis.
– We present preliminary results from applying our implementation (see `https://github.com/saffriha/ictac2014`) to a range of benchmarks.

1.2 Outline

Section 2 informally presents the GAG-based side-effect analysis and its supporting analyses, in particular, points-to analysis. Section 3 formally defines the domain of generalized access graphs and its operations and establishes its basic properties. Section 4 specifies the intraprocedural points-to analysis and Section 6 extends it to a side-effect analysis. Section 7 sketches the interprocedural analysis. Section 8 reports on our experiments with the implementation. Section 9 discusses related work and Section 10 concludes.

2 Side-Effect Analysis

This section presents motivational examples that demonstrate various aspects of our side-effect analysis and in particular uses of GAGs in method summaries.

Our running example concerns some methods written for objects of class `List`, which represent a node in a linked list with integer elements.

```
class List {
  int v;
  List n;
  List(int v, List n) {
    this.v = v;
    this.n = n;
  }
}
```

2.1 Simple Method

The method `foo` takes a list as its argument. It modifies the list node and returns the rest of the argument list.

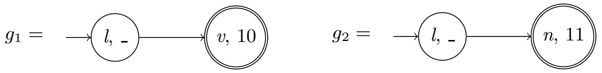


Fig. 1. Write effect and abstract return object of `foo`

```
 9 List foo(List l) {
10   l.v = 0;
11   return l.n;
12 }
```

The GAG g_1 in Fig. 1 describes the write effect of the method whereas g_2 describes the returned value (and also the read effect). Both specify the outcome in isolation, that is, without regard for the calling context of the method.

Each path through a GAG from a root node to an accepting node corresponds to a potential access path that starts from any object that may be supplied for the root node later on. Each identifier on this path is interpreted as a component of the access path. In general, a GAG may represent any number of paths.

In this case, the root node is l in both graphs. It stands for the formal parameter of the method. Further, each graph represents exactly one path, g_1 represents the path $l.v$ whereas g_2 represents $l.n$.

The GAG also includes the program point of each field access in its nodes. The example uses line numbers to indicate program points. If the program points are important, then we write access paths with program points as superscripts as in $l.v^{10}$. These program point annotations play a crucial role in effectively finding a fixpoint during the analysis.

Thus, the write effect of `foo` consists of the potential modification of $l.v$ whereas the returned object is described by $l.n$. That is, the GAG describes the set of parameter rooted SAPs written by the method. The GAG computed by the analysis yields access paths that are valid in the pre-state of the method, that is, in the program state at invocation time of the method.

The return value of `foo` is described by g_2 as the abstract object that can be reached via the path $l.n^{11}$. Again, the superscript 11 indicates the program point containing the access to the field n.

2.2 Loops

So far we have seen how GAGs represent summaries of the side effects and the return value of a simple method. Next, we put the body of `foo` in a loop to observe summarization at work. The comments in the listing contain the abstract object bound to r after analyzing the line in regular expression notation.

```
13 List loop(List l) {     // round 1 // round 2  // round 3
14   List r = l;           // r ↦ l
15   while (r.v != 42) {  //          // r ↦ l.n? // r ↦ l.n*
16     r.v = 0;
17     r = r.n;            // r ↦ l.n // r ↦ l.n+ // r ↦ l.n+
18   }                                            // r ↦ l.n*
19   return r;
20 }
```

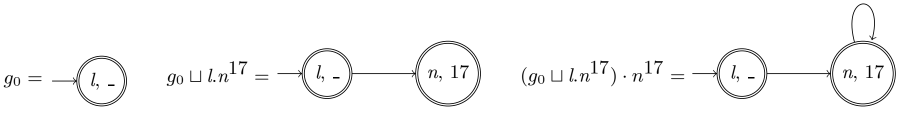


Fig. 2. GAGs to analyze `loop`

Before entering line 15, our analysis finds that r points to l. This fact is represented by an environment that maps r to the abstract object represented by the GAG g_0 (Fig. 2). After line 17, unsurprisingly, r points to the object represented by $l.n^{17}$. Then the body of the while loop is reanalyzed with r bound to the join of g_0 and $l.n^{17}$. By the end of the loop, r is updated to $(g_0 \sqcup l.n^{17}) \cdot n^{17}$, which does not change anymore. This fixpoint is reached because each access at a particular program point is represented at most once in an access graph. See Section 3 for the formal definition of the operators. Thus, the return value corresponds to the path set generated by the regular expression $l.n^*$. Similarly, the side effect may be computed as $l.n^*.v$ (see Fig. 4).

2.3 Method Calls

Next, we change the program to call the method `foo` in the loop.

```
21 List loopCall(List l) {
22   List r = l;
23   while (r != null)
24     r = foo(r);
25   return r;
26 }
```

Up to line 24, r is bound to g_0 as before. Analyzing the method call just fetches the return value g_2 from `foo`'s method summary. In general, this value is phrased in terms of the formal parameters of the callee, so that it must be translated to the caller. In this case, the translation replaces l by l, so the body of the while loop is reanalyzed with r bound to $g_0 \sqcup g_2$ (Fig. 3).

Reanalyzing the call yields the same return value g_2, but now the parameter value $g_0 \sqcup g_2$ must be substituted for l in g_2, which yields $(g_0 \sqcup g_2) \cdot n^{11}$ (Fig. 3) representing the same access paths as $(g_0 \sqcup l.n^{17}) \cdot n^{17}$ in the analysis of *loop*.

The write effect is computed in a similar way. In the first analysis of the method call, we replace l with l in `foo`'s summary resulting in a write effect of $l.v^{10}$. In the next iteration, $g_0 \sqcup g_2$ is substituted for the formal parameter in `foo`'s summary. The resulting write effect is $l.(n^{11})?.v^{10}$. The abstract object resulting from the return value is $l.(n^{11})^+$. As seen in Fig. 2, the concatenation of a field access at the same program point results in a loop on the GAG node representing this field access. The new environment entry for r is $r \mapsto l.(n^{11})^*$ with the resulting write effect $l.(n^{11})^*.v^{10}$.

The next iteration reaches the fixpoint. The method summary for *loopCall* consists of the abstract object from r's environment entry $l.(n^{11})^*$ and the write effect $l.(n^{11})^*.v^{10}$ (see Fig. 4).

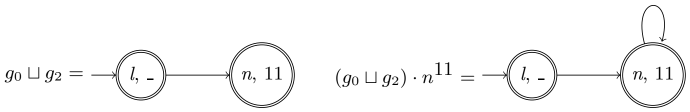


Fig. 3. GAGs to analyze `loopCall`

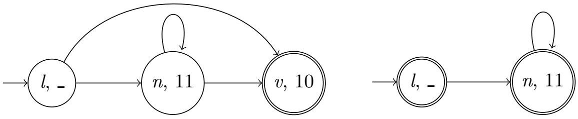


Fig. 4. `loopCall`'s write effects (left) and abstract return object (right)

2.4 New Object Abstraction

The next examples deal with allocation and aliasing. They unveil some further information that is computed by the analysis: points-to and escape information. To simplify the presentation, we ignore side effects in the following examples. Thus, for the rest of this section we only regard two components of a method summary:

1. may-points-to information that describes aliases created by the method and
2. the abstract object returned by the method.

First, we turn to the representation of new objects. Here is a method that constructs and attaches a new element to a list.

```
27 List newList(int v, List n) {
28   assert (v < 10);
29   List l = new List(v, n);
30   return l;
31 }
```

The constructor call on line 29 creates a points-to entry $\texttt{new}^{29}.n^6 \to n$ (with the 6 coming from the `List` constructor) and the assignment on line 29 creates an environment entry for l of $\texttt{new}^{29}$. Thus, the method summary consists of the points-to relation $\{\texttt{new}^{29}.n^6 \to n\}$ and the abstract return object $\texttt{new}^{29}$, which indicates that this object may escape.

Given the method summary, we consider a method that calls `newList` in a loop.

```
32 List newNList() {
33   int i = 0;
34   List r = null;
35   while (i < 10) {
36     r = newList(i, r);
37     i++;
38   }
39   return r;
40 }
```

The points-to relation in the method summary of `newNList` is $\texttt{new}^{29}.n^{6} \to \texttt{new}^{29}$ and the abstract return object is $\texttt{new}^{29}$. We do not explicitly represent `null` in the abstract domain as we only consider may-points-to information.

2.5 Aliasing

Another group of examples demonstrates how aliasing is described in terms of the pre-state of the method. We omit program points and use the symbol • as the rightmost field selection operator in SAPs describing references. The following method swaps the first two entries of the passed list and returns the resulting list.

```
List swap(List l) {
  List ln = l.n;
  List lnn = ln.n;
  l.n = lnn;
  ln.n = l;
  return ln;
}
```

The method's points-to summary is $l \bullet n \to l.n.n$, $l.n \bullet n \to l$. The GAG $l.n$ describes the returned value. This result demonstrates that the summary consists of paths that refer to the pre-state of the method.

A final aliasing example illustrates the interaction between the information from the method summary and the points-to set at the call site. An auxiliary method `depth2` overwrites the n field two elements down the list. Method `depth1` overwrites the n field of the first element and then calls `depth2`.

```
48 void depth2 (List x, List y) {
49   List v = y.n;      // env: v ↦ y.n
50   v.n = x;           // points-to: {(y.n) • n → x}
51 }
52 void depth1 (List x, List y, List z) {
53   y.n = z;           // points-to: {y • n → z}
54   depth2 (x, y);     // points-to: {y • n → z, (y.n) • n → x, z • n → x}
55 }
```

The annotations in the listing state the intermediate results of the analysis after executing the statement on the line. The annotation `env` states the binding of v and `points-to` states the accumulated points-to set up to that point. Most annotations are straightforward, except the points-to set on line 54 after the call to `depth2`. The first entry, $y \bullet n \to z$, is carried through from the previous statement. The second entry, $(y.n) \bullet n \to x$, is obtained by substituting the abstraction of the formal parameters in the points-to summary of `depth2` from line 50. In this case, the substitution is the identity. The last entry, $z \bullet n \to x$, is generated by our resolution algorithm that integrates the points-to information at the call site with the method summary. The abstract object $y.n$ in the summary interacts with the first points-to entry that states that $y \bullet n$ may also point to z. Thus, the last entry results from contracting $y.n$ to z in the summary. No entry can be removed because the points-to information is not definitive (may-points-to information).

2.6 Global Variables

A final example demonstrates side effects on global variables.

```
void setGlobal(List l) {
  Global.g = l;
}
```

The points-to relation in `setGlobal`'s summary is $\mathit{Global} \bullet g \to l$.

3 Abstract Domain: Generalized Access Graphs

A generalized access graph represents a possibly infinite set of access paths relative to a set of roots. These roots are usually abstract objects like method parameters or allocation points.

Definition 1. *An* occurrence *of an identifier or an allocation in a program P is specified with an element from $\mathit{Occ}_P = (\mathit{ID}_P \uplus \{\texttt{new}\}) \times \mathit{PP}_P$ where ID_P is the set of identifiers occurring in P and PP_P is the set of program points of P. We write $\texttt{C}.x \in \mathit{ID}_P$ to refer to static global variable x of class* C*.*

In the rest of this section, we take for granted that all definitions are relative to an arbitrary, fixed program P and thus drop the $_P$ subscript.

Definition 2. *A* generalized access graph *for program P is a tuple $\langle N, E, A, R\rangle$*

- *$N \subseteq \mathit{Occ}$ is a set of identifiers or allocation occurrences of P,*
- *$E \subseteq N \times N$ is a set of directed edges,*
- *$A \subseteq N$ is a set of* accepting nodes*, and*
- *$R \subseteq N$ is a set of* root nodes*.*

If g is a generalized access graph, then we sometimes write $N(g)$, $E(g)$, $A(g)$, and $R(g)$ for its components.

As Occ is finite, the set GAG of all generalized access graphs for P is also finite. We write $\mathit{id}(n)$ and $\mathit{pp}(n)$ to extract the identifier and program point of a node $n \in N$.

Khedker et al. proposed a closely related notion of access graphs [15], which we generalize in several respects: we allow non-accepting nodes, we allow several root nodes instead of just a single root node, and we do not distinguish between "normal" access graphs and "remainder" graphs as they do.

We use two notations, a graphical one and another based on regular expressions, to concisely write generalized access graphs. Both are inspired by the close relationship to nondeterministic finite automata. Consider the access graph with $N = \{\langle x, 1\rangle, \ \langle f, 2\rangle, \ \langle f, 3\rangle\}$, $E = \{\langle\langle x, 1\rangle, \langle f, 2\rangle\rangle, \ \langle\langle f, 2\rangle, \langle f, 2\rangle\rangle, \ \langle\langle f, 2\rangle, \langle f, 3\rangle\rangle\}$, $A = \{\langle f, 3\rangle\}$, and $R = \{\langle x, 1\rangle\}$. Its regular expression notation is $x^1.f^2\texttt{+}.f^3$ and Fig. 5 shows its graphical representation.

Definition 3. *Let $g \in \mathit{GAG}$ be an access graph. The* indexed path language *$L^p(g) \subseteq \mathit{Occ}^*$ of g is defined as the language of the nondeterministic finite automaton $\langle N \uplus \{q_0\}, \mathit{Occ}, \delta, \{q_0\}, A\rangle$ where δ is the smallest relation such that*

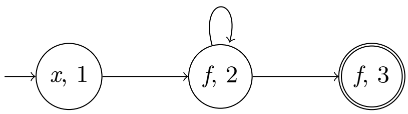
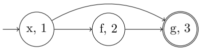
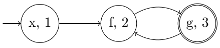


Fig. 5. Example access graph

- $(q_0, \langle x,p\rangle, \langle x,p\rangle) \in \delta$, *for each* $\langle x,p\rangle \in R(g)$,
- $(n, \langle x,p\rangle, \langle x,p\rangle) \in \delta$, *for each* $\langle n, \langle x,p\rangle\rangle \in E(g)$.

The path language *of* g *is* $L(g) = \{x_1 \ldots x_n \mid \langle x_1,p_1\rangle \ldots \langle x_n,p_n\rangle \in L^p(g)\}$.

Here are two examples:

$L($  $) = \{x.g,\ x.f.g\}$ $L($  $) = x.(f.g)$+

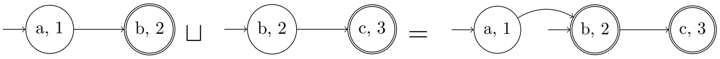
Lemma 1. *Let* g *be an access graph. The path language of* g, $L(g)$, *is regular.*

Next, we define a join operation on access graphs that approximates the union of their path languages.

Definition 4. *For* $i \in \{1,2\}$, *let* $g_i = \langle N_i, E_i, A_i, R_i\rangle$ *be access graphs. Define their* join $g_1 \sqcup g_2 = \langle N_1 \cup N_2, E_1 \cup E_2, A_1 \cup A_2, R_1 \cup R_2\rangle$.

Lemma 2. $L(g_1 \sqcup g_2) \supseteq L(g_1) \cup L(g_2)$.

In general, $L(g_1 \sqcup g_2)$ may contain words that are neither in $L(g_1)$ nor in $L(g_2)$ because their underlying node sets need not be disjoint. For example, consider



where the language of the joined graph contains the word $a.b.c$ which is not in the language of either argument graph: $\{a.b\}$ and $\{b.c\}$, respectively.

Theorem 1. *For each program* P, *the structure* $\langle GAG, \sqcup, \sqcap, \bot, \top\rangle$ *is a finite, complete lattice. The meet operation* $\sqcap$ *is componentwise intersection,* $\bot = \langle\emptyset,\emptyset,\emptyset,\emptyset\rangle$, *and* $\top = \langle N, N \times N, N, N\rangle$ *where* $N = Occ \times PP$.

The lattice ordering, which corresponds to the componentwise subset ordering, is defined by $g_1 \sqsubseteq g_2$ iff $g_1 \sqcup g_2 = g_2$. Furthermore, $g_1 \sqsubseteq g_2$ implies $L(g_1) \subseteq L(g_2)$.

Remark 1. Our analysis does not make use of the meet operation, but it is easy to see that $L(g_1 \sqcap g_2) \subseteq L(g_1) \cap L(g_2)$: Suppose there is a path in $g_1 \sqcap g_2$ from a root node $r \in R(g_1) \cap R(g_2)$ to an accepting node $a \in A(g_1) \cap A(g_2)$. Then this path exists in g_1 and g_2, too, as it exists in $E(g_1) \cap E(g_2)$.

As an example that the inclusion is proper consider $g_1 = a^1$ and $g_2 = a^2$. It holds that $L(g_1 \sqcap g_2) = \emptyset$ but $L(g_1) \cap L(g_2) \neq \emptyset$.

The analysis requires one more operation. Concatenation of access graphs computes an approximation to the concatenation of their languages.

Definition 5. *For $i \in \{1, 2\}$, let $g_i = \langle N_i, E_i, A_i, R_i\rangle$ be access graphs. Define their* concatenation $g_1 \cdot g_2 = \langle N_1 \cup N_2, E_1 \cup E_2 \cup (A_1 \times R_2), A_2, R_1\rangle$.

Lemma 3. $L(g_1 \cdot g_2) \supseteq L(g_1) \cdot L(g_2)$.

As an example that the inclusion may be proper, consider $L(a^1) = \{a\}$ and $L(a^1 \cdot a^1) = a^+ \supsetneq \{a.a\}$.

Lemma 4. *Concatenation is monotone in both arguments.*

4 Intraprocedural Points-to Analysis

The first step towards our side-effect analysis is a points-to analysis for an imperative core language with objects, which is the essence of an intermediate representation for Java like Jimple. We consider a program one method at a time and we assume that each method is given in the form of a control-flow graph $CFG = (V, F)$ where the nodes $V = PP$ correspond to program points and the directed edges $F \subseteq V \times V$ correspond to potential control transfers between program points. The function $pred : V \to \mathcal{P}(V)$ maps a node to its set of predecessors: $pred(v) = \{v' \mid (v', v) \in F\}$. There are two distinct nodes that determine the entry point and the exit from the method.

Each node v in the CFG is associated with a statement $stm(v)$ of one of the following forms, where $x, y, \ldots$ range over local variables.

- $x = y \oplus z$, primitive operation;
- $x = c$, constant including `null`;
- $x = y$, copy;
- $x = \texttt{new}^p$, allocate a new uninitialized object of type `T`;
- $x = y.a$, read field a from object y;
- $x.a = y$, write field a in object x;
- $x = \texttt{call}\ f(y_1, \ldots, y_n)$, call method f.

4.1 Memory Abstraction

An abstract object is described by a generalized access graph. We use a different symbol to emphasize the interpretation of the graph as an abstract object.

$$o \in Obj = GAG$$

The object o represents the set of objects that can be reached in the pre-state of a method call via the access paths in $L(o)$. The graphs are anchored either in the formal parameters of the method, in global variables that can be accessed inside the method, or in objects newly allocated during the method call. Without loss of generality, we pretend that all such allocations take place before the method starts, so that they are representable in the pre-state already.

To represent points-to information, we need references to fields of abstract objects. Such a reference is a pair of an abstract object and a field name.

$$Ref \ni r ::= o \bullet a$$

Points-to information itself is represented by a points-to set of the form $P \in \mathcal{P}(Ref \times Obj)$. An element $\langle r, o \rangle \in P$ states that the reference r *may* point to abstracted object o. In addition, each reference $o \bullet a$ points to an implicit natural object, namely $o \cdot a$, which implements the may-nature of the analysis.

In principle, we might also represent points-to information with a partial mapping $Ref \hookrightarrow Obj$ by joining multiple target objects for the same reference. By keeping a set of target objects, we retain some more precision. We have yet to investigate whether it makes a difference in practice.

4.2 Method Summaries

We assume that, for each method, there is a method summary that describes the result of a method call and the potential side effects of the method. Specifically, for a method f

– $returns(f) \in Obj$ describes the return value of the method as an abstract object in terms of f's formal parameters;
– $exitSet(f) \in \mathcal{P}(Ref \times Obj)$ describes the potential modification of the points-to information by calling f;
– $reads(f) \in GAG$ describes the set of objects that may be read during execution of f;
– $writes(f) \in GAG$ describes the set of objects that may be written to during execution of f.

These functions do not take into account aliasing that is present at a call site of the method f. Thus, the method summary needs to be adapted to the circumstances at each call site. On the positive side, it means that our analysis is modular, because after generating the method summary for f, all further analysis can rely on the summary.

4.3 Dataflow Equations

The domain DP of the dataflow analysis consists of a local variable environment that maps a variable name to an abstract memory location and a points-to relation P as described above. We model the environment ρ as a partial map that we consider as a set of pairs when convenient: $\rho \in Env = Var \hookrightarrow Obj$.

$$DP = Env \times \mathcal{P}(Ref \times Obj)$$

The dataflow equations for the points-to analysis are typical for a forward analysis. For each node v in the CFG, they determine values $inP(v), outP(v) \in DP$ that accumulate the analysis result and there are functions $genP(v), killP(v) \in DP$

that compute information to add to or remove from an intermediate result. As a slight difference to the standard framework, the value of $genP(v)$ often depends on $inP(v)$.

$$inP(v) = \bigsqcup_{p \in pred(v)} outP(p) \qquad outP(v) = (inP(v) - killP(v)) \sqcup genP(v) \tag{1}$$

The join operation on the environment is the pointwise join of the abstract objects in the range. It is set union on the points-to sets. The "−" operation computes the set difference on the underlying sets. The initial state is given by $inP(v) = outP(v) = (\emptyset, \emptyset)$, with one exception:

$$outP(entry) = ([x \mapsto x^p \mid x \text{ formal parameter defined at program point } p], \emptyset)$$

The result of the analysis is the least fixpoint of the equations (1).

The $genP$ and $killP$ functions applied to node v are defined by case analysis on the statement at node v. If $stm(v)$ has the form $x = \ldots$, then $killP(v) = (\{(x, o) \mid o \in Obj\}, \emptyset)$, that is, the previous assignment to x is removed. For other forms of statements, we specify the kill set explicitly.

Generally, let $(\rho_{in}, P_{in}) = inP(v)$ in the following definition of $genP$ and $killP$.

- If $stm(v)$ is $x = y \oplus z$ or $x = c$, then $genP(v) = (\emptyset, \emptyset)$.
- If $stm(v)$ is $x = y$, then $genP(v) = ([x \mapsto \rho_{in}(y)], \emptyset)$.
- If $stm(v)$ is $x = \mathtt{new}^p$, then $genP(v) = ([x \mapsto \ell^p], \emptyset)$, so that x points to an abstract object allocated at program point p.
- If $stm(v)$ is $x = y.a^p$, then $genP(v) = ([x \mapsto \bigsqcup objs(inP(v), y.a^p)], \emptyset)$. The function $objs : DP \times Var \times Field \times PP \to \mathcal{P}(Obj)$ resolves a field access under a given points-to set.

$$objs((\rho, P), y.a^p) = \{\rho(y) \cdot a^p\} \cup \{o' \mid (o \bullet b, o') \in P, mayAlias(o, \rho(y)), a = b\}$$

The first part concerns the direct effect of the field access. It concatenates the abstract object that is currently stored in the variable with the trivial access graph to field a^p. The second part is the indirect effect. If the points-to set contains evidence that $y.a$ may also point to some object o', then that object is also a potential result.

Checking the last part is more involved than comparing o and $\rho(y)$ for equality. As both are represented by method-local access graphs, it may be the case that they are not equal but nevertheless have some access paths in common. The function $mayAlias$ checks the absence of such common access paths by checking disjointness of the path languages:

$$mayAlias(o_1, o_2) = L(o_1) \cap L(o_2) \neq \emptyset$$

As the path languages are regular (cf. Lemma 1), this check is effective. If we consider the booleans ordered by $\mathtt{false} \sqsubseteq \mathtt{true}$, then $mayAlias$ is monotone in both arguments and thus $objs$ is also monotone in ρ and P.

- If $\mathit{stm}(v)$ is $x.a^p = y$, then $\mathit{killP}(v) = (\emptyset, \emptyset)$ because no local (or global) variable is overwritten but after the assignment $x.a$ may point to the object stored in y which must be reflected in the points-to information. Hence, $\mathit{genP}(v) = (\emptyset, \mathit{refs}(\rho_{in}, x.a^p) \times \{\rho_{in}(y)\})$. The function $\mathit{refs} : \mathit{Env} \times \mathit{Var} \times \mathit{Field} \times \mathit{PP} \to \mathit{Ref}$ resolves a field access to a reference, a symbolic left value.

$$\mathit{refs}(\rho, x.a^p) = \rho(x) \bullet a$$

- If $\mathit{stm}(v)$ is $x = \texttt{call}\ f(y_1, \ldots, y_k)$, then we first need to consult the call graph for the set of possible call targets $f_1, \ldots, f_t$. The gen-information of the method call is joined from the individual call targets: $\mathit{genP}(v) = \bigsqcup_j d_j$. For each target f_j with formal parameters $x_1, \ldots, x_k$, define $d_j = ([x \mapsto o'_j], P'_j)$ where o'_j describes the abstract object returned by the method call and P'_j describes the potential side-effect of the call on the parameters and global variables.

  We obtain this data from the method summary of f_j, but as this summary contains information that is local to f_j, it needs to be translated to the calling context. In particular, the access graphs in f_j's summary refer to the x_i, the names of f_j's formal parameters. They need to be replaced by the abstract objects $\rho_{in}(y_i)$, for $1 \leq i \leq k$, representing the parameters of the call site.

  However, this replacement alone is not sufficient, because the access paths in the result ignore aliasing (points-to information) that is present at the call site: a method is always analyzed under the assumption that its arguments are not aliased. This discrepancy has to be corrected by retracing the access paths in $\mathit{returns}(f_j)$ using the points-to information at the call site, which is represented by P_{in}. Thus, if $o = \mathit{returns}(f_j)$ then

$$o'_j = \mathit{trans}_o(\rho_{in}, P_{in}, o) = \bigsqcup_{n \in A(o)} Q(o, n)$$

  where $Q : \mathit{Obj} \times \mathit{Occ} \to \mathit{Obj}$ is the smallest function such that

$$\begin{aligned} Q(o, \langle \texttt{new}, p\rangle) &\sqsupseteq \ell^p \quad \text{if } \langle \texttt{new}, p\rangle \in R(o) \\ Q(o, \langle x_i, p\rangle) &\sqsupseteq \rho_{in}(y_i) \quad \text{if } \langle x_i, p\rangle \in R(o) \qquad \text{substituting formal parameter} \\ Q(o, \langle a, p\rangle) &\sqsupseteq Q(o, n) \cdot a^p \sqcup \bigsqcup \{t \mid \langle o' \bullet a, t\rangle \in P_{in}, \mathit{mayAlias}(o', Q(o, n))\} \\ &\quad \text{for all } n \text{ such that } \langle n, \langle a, p\rangle\rangle \in E(o) \end{aligned}$$

  A similar transformation has to be applied to the points-to set returned by the method.

$$P'_j = \{\langle \mathit{trans}_r(\rho_{in}, P_{in}, r), \mathit{trans}_o(\rho_{in}, P_{in}, o)\rangle \mid \langle r, o\rangle \in \mathit{exitSet}(f)\}$$

  where

$$\mathit{trans}_r(\rho, P, o \bullet a) = \mathit{trans}_o(\rho, P, o) \bullet a$$

5 Global Variables

Global variables are straightforward to integrate into the analysis. The environment ρ also maintains information about the abstract objects contained in the global variables. That is, the initial environment in $\mathit{outP}(\mathit{entry})$ also contains bindings $[\mathtt{C}.a \mapsto \mathtt{C}.a^p]$, where p is the program point defining a in $\mathtt{C}$.

There are two new cases for $\mathit{genP}(v)$ where $(\rho_{in}, P_{in}) = \mathit{inP}(v)$ and the method call needs to be extended.

- If $\mathit{stm}(v)$ is $x = \mathtt{C}.a$, then $\mathit{genP}(v) = ([x \mapsto \rho_{in}(\mathtt{C}.a)], \emptyset)$.
- If $\mathit{stm}(v)$ is $\mathtt{C}.a^p = y$, then $\mathit{killP}(v) = (\{(\mathtt{C}.a, o) \mid o \in \mathit{Obj}\}, \emptyset)$ and $\mathit{genP}(v) = ([\mathtt{C}.a \mapsto \rho_{in}(y)], \emptyset)$.
- If $\mathit{stm}(v)$ is $x = \mathtt{call}\ f(y_1, \ldots, y_k)$, then we extend the previous treatment. Let $\rho_{out} = \mathit{globals}(f)$ be the environment at the exit node of f restricted to the bindings of the global variables (also part of the method summary). Then the environment part of $\mathit{genP}(v)$ needs to be extended with $[\mathtt{C}.a \mapsto \mathit{trans}_o(\rho, P, \rho_{out}(\mathtt{C}.a))]$ for each global variable $\mathtt{C}.a$.[1]

  To transfer these entries successfully, we need to extend the Q function in the definition of trans_o by

  $$Q(o, \langle \mathtt{C}.a, p\rangle) \sqsupseteq \rho_{in}(\mathtt{C}.a) \quad \text{if } \langle \mathtt{C}.a, p\rangle \in R(o)$$

The treatment of function calls could be improved by additionally keeping track of which global variables are *definitely* overwritten by the call. The entries for these variables could be killed from the environment and replaced by the information from the method summary.

6 Intraprocedural Side-Effect Analysis

To perform the side-effect analysis, we assume that the results of the points-to analysis are available in $\mathit{inP}(v)$ and $\mathit{outP}(v)$, for each CFG node v. The domain for this analysis is the product lattice of two access graphs, the first one summarizing read accesses, the second one write accesses.

$$\mathit{DS} = \mathit{GAG} \times \mathit{GAG}$$

The analysis is again a forward analysis, but in this case there are no kill sets.

$$\mathit{inS}(v) = \bigsqcup_{p \in \mathit{pred}(v)} \mathit{outS}(p) \qquad \mathit{outS}(v) = \mathit{inS}(v) \sqcup \mathit{genS}(v) \tag{2}$$

All values are initialized to the bottom of the lattice $\mathit{inS}(v) = \mathit{outS}(v) = (\bot, \bot)$. The result of the analysis is the least fixpoint of the equations (2).

Again, we define $\mathit{genS}(v)$ by cases on the statement at node v. Let $(\rho_{in}, P_{in}) = \mathit{inP}(v)$ be the result of the points-to analysis at node v.

[1] In an implementation, it is sufficient to only keep entries for those variables that are actually used inside f.

```
InterProceduralAnalysis()
Compute call graph and its SCC tree.
for each method $f$
    $summaries[f] = (\bot, \emptyset, \bot, \bot)$
while an unprocessed SCC exists
    Choose an unprocessed SCC $S$ where all predecessors are processed.
    repeat
        $done = true$
        for each method $f$ in $S$
            $newSummary$ = IntraProceduralAnalysis($f, summaries$)
            if $summaries[f] \neq newSummary$
                $summaries[f] = newSummary$
                $done = false$
    until $done$
    Mark $S$ as processed.
```

Fig. 6. Algorithm for interprocedural analysis

- If $stm(v)$ is $x = y \oplus z$ or $x = c$ or $x = y$ or $x = \texttt{new}^p$, then $genS(v) = (\emptyset, \emptyset)$.
- If $stm(v)$ is $x = y.a^p$, then $genS(v) = (\rho_{in}(y) \cdot a^p, \emptyset)$.
- If $stm(v)$ is $x.a^p = y$, then $genS(v) = (\emptyset, \rho_{in}(x) \cdot a^p)$.
- If $stm(v)$ is $x = \texttt{call}\ f(y_1, \ldots, y_n)$, then $genS(v) = (g_r, g_w)$ where

  $$g_r = trans_o(\rho_{in}, P_{in}, reads(f)) \qquad g_w = trans_o(\rho_{in}, P_{in}, writes(f))$$

  As in the points-to analysis, the method summary needs to be translated into the current environment and aliasing context, and we need to join the information of the possible call targets for f.

Allocations are not registered as write effects because they do not modify existing data structures. However, reads and writes to newly allocated data appear as side-effects. No special treatment is needed to cater for global variables.

7 Interprocedural Analysis

The interprocedural analysis computes the method summaries for the whole program by repeatedly applying the intraprocedural analysis to the program's functions until a fixpoint is reached. We sketch the algorithm in Fig. 6.

At first, the program's call graph and its strongly connected components (SCCs) are computed. Next, the SCCs are traversed bottom-up and for each SCC the fixpoints of its methods' summaries are computed starting from the bottom values of the respective domains.

The fixpoint computation recomputes the method summaries for all methods contained in the current SCC. This computation is repeated until all summaries stabilize. If any method summary changes, then all summaries in the current SCC have to be recomputed because they mutually depend on each other.

8 Experience

We implemented both the points-to analysis and side-effect analysis on top of the SOOT Java bytecode analysis and transformation framework in version 2.5.0. To increase the scalability of our analysis points-to pairs with structurally equal left hand sides are joined into a single pair.

Evaluation. The evaluation concentrates on analysis time and precision of our analyses. We focus on relatively small benchmark programs from the JOlden [3] benchmark suite. The suite consists of ten benchmark programs.

We stripped the benchmarks of the time measurement, statistics and printing functionality that is common to all programs in the JOlden suite to avoid analyzing large parts of the JDK. For example, the unstripped version of the Bisort benchmark had a call graph containing more than 8000 methods although Bisort's functionality is implemented in 11 user methods. All benchmarks were executed on a machine with AMD Phenom II X6 (2.8 GHz, 6 Cores) processor and 8 GB RAM on top of Archlinux 64bit, Kernel 3.13.8-1 and OpenJDK 7.0.

We present the results of running our points-to and side-effect analysis on nine of the JOlden benchmarks in Table 1. In each case, the analysis processed all methods, user and library, that are reachable from the main method. We excluded static initializers and external methods (methods without an active body) from the analysis. For each application, we present the total number of methods (including library methods) and the number of user methods. We measured the total run time and the run time of the call graph construction including the calculation of the SCCs. As a quality measure that is independent from our abstract domain we count the methods that do not introduce new aliases and the pure methods. Here, "pure" means that a method does not have any write effects on heap-allocated memory locations in the prestate of the method. Following the JML convention, we consider constructors that only mutate fields of "this" as pure.

For the benchmarks marked with * we excluded the JDK methods from the call graph, as our prototype apparently does not scale to the thousands of library methods that can be transitively invoked by these benchmarks.

Discussion. The run time of the analysis is within reasonable bounds for small programs, generally running in a fraction of the time taken for the call graph construction and finding the SCCs. We still need to gain experience with larger programs.

For the benchmark programs that we have analyzed our analysis gives useful results with a precision that is roughly comparable to that of others [21]. In the best case `TreeAdd` we can identify roughly 89% of the methods as pure. The other extreme is `MST` where we identify 21% of the methods as pure.

Table 1. Analysis results for the Java Olden benchmarks

Application	Methods		Run time			Method summaries	
	User	All	CG+SCC	Analysis	Total	No new aliases	Pure
BH	53	68	57.26s	1.17s	58.43s	76.47%	66.18%
BiSort	11	13	5.40s	1.05s	6.45s	61.54%	38.46%
Em3d*	16	16	54.50s	0.90s	55.40s	50.00%	37.50%
MST	29	34	55.45s	0.83s	56.28s	85.29%	20.59%
Perimeter	34	36	5.75s	0.95s	6.70s	94.44%	80.56%
Power	26	34	55.65s	2.81s	58.46s	88.24%	50.00%
TreeAdd	3	9	52.67s	0.45s	53.12s	88.89%	88.89%
TSP*	12	12	52.05s	1.03s	53.08s	66.67%	33.34%
Voronoi*	55	55	58.15s	1.69s	59.84s	89.09%	76.36%

9 Related Work

There is a plethora of literature on effect and points-to analysis for heap-allocated objects. We therefore focus on the distinguishing features of our abstract domain and compare our work to selected bottom-up points-to, shape and effect analyses.

Regarding our abstract domain, the most closely related work is by Khedker et al. [15]. They have introduced access graphs, which include program points in their nodes to deal with unboundedly large data structures. We extend access graphs to GAGs, which facilitate a compact representation of the points-to relation that cannot be achieved with (sets of) simple access graphs.

While Khedker et al. rely on access graphs for a number of analyses including alias analysis, they do not employ them for points-to analysis, as we do. In contrast to our abstract domain, they use partly "unresolved" access graphs to represent abstract references in their alias sets. That is, their access graphs can be rooted in a reference that requires further resolution to obtain the abstract object it stands for, whereas we use "resolved" (up to the unknown context) abstract objects being rooted in parameters or allocation points. We use such resolved GAGs, as they avoid repeated resolution of the same access graphs while preserving precision on updates to references, which we consider as an advantage in our flow-sensitive analysis.

Most of the following proposals share the property with ours that their abstract domains are based on Larus' *access paths* [18] or Deutsch's SAPs [8].

All data-flow algorithms must deal with the unbounded nature of recursive data structures. Many proposals [17,4,6] follow the *k-limiting* approach [12], which limits access paths by truncation.

While our proposal uses a *storeless model* (originally proposed by Jonkers [13]) other proposals [18,21] use a *store-based model* model employing some form of (rooted) directed graph representation or a compact representation thereof [11] for points-to or alias information. A store-based model can enable the description of regular patterns across references and the objects these references refer to.

In their side-effect analysis for Java, Sălcianu and Rinard [21] represent the points-to relation as a *points-to graph*, a rooted directed graph representation.

Their points-to graphs allow multiple root nodes (as our GAGs do), but do not include program points in their points-to graphs. Larus and Hilfinger's *alias graphs* [18] are similar to Sălcianu's points-to graphs.

Several authors [14,24] propose scalable bottom-up pointer analyses for C programs, but ignore heap-allocated data. Matosevic et al. [19] use a SAP-based abstract domain in their bottom-up side-effect analysis, but their loop abstraction mechanism can only detect three patterns of iteration. Moreover, they do not formally describe how they handle method calls. In contrast to our points-to relation, which can be viewed as a *total transfer function*, their abstract domain serves as a *partial transfer function* [20] that assumes the context to have certain properties. Partial transfer functions can be considered as an optimization that is also applicable to our analysis. Gulavani et al. [10] propose a bottom-up shape-analysis based on separation logic. Their *Logic of Iterated Separation Formulae* allows the computation of a loop summary from a loop body summary.

Another widely-used and highly scalable proposal for points-to analysis is Steensgaard's [22] type-based analysis using unification, which is most suitable for statically-typed languages. Our analysis can be combined with type-based information to improve both precision and scalability.

10 Conclusion

Side-effect analysis is an important tool in the programmer's toolbox. It aids program understanding, supports other program analyses, and it enables advanced program optimizations and safe parallel execution. Our analysis is based on a single comprehensive and precise abstract domain of generalized access graphs, which serve to express abstract objects, points-to information, escape information, as well as read and write effects. In our experience, the single abstract domain simplifies the implementation. The preliminary data gathered from our implementation shows that our approach is practically feasible, but we believe that further algorithmic tuning is possible.

References

1. Bocchino Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In: Arora, S., Leavens, G.T. (eds.) OOPSLA, pp. 97–116. ACM (2009)
2. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. 7(3), 212–232 (2005)
3. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in java. In: IEEE PACT, pp. 280–291. IEEE Computer Society (2001)
4. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: Adsul, B., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 172–186. Springer, Heidelberg (2007)

5. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) ICSE, Limerick, Ireland, pp. 439–448. ACM (June 2000)
6. Dasgupta, S., Karkare, A., Reddy, V.K.: Precise shape analysis using field sensitivity. ISSE 9(2), 79–93 (2013)
7. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
8. Deutsch, A.: Interprocedural alias analysis for pointers: Beyond k-limiting. In: Sarkar, V., Ryder, B.G., Soffa, M.L. (eds.) PLDI, pp. 230–241. ACM (1994)
9. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Knoop, J., Hendren, L.J. (eds.) PLDI, Berlin, Germany, pp. 234–245. ACM Press (2002)
10. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis using lisf. ACM Trans. Program. Lang. Syst. 33(5), 17 (2011)
11. Hind, M., Burke, M.G., Carini, P.R., Choi, J.-D.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21(4), 848–894 (1999)
12. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Proc. of the 9th ACM Symp. POPL, Albuquerque, New Mexico, USA, pp. 66–74. ACM Press (1982)
13. Jonkers, H.B.M.: Abstract storage structures. In: de Bakker, van Vllet (eds.) Algorithmic Languages. IFIP, pp. 321–343 (1981)
14. Kang, H.-G., Han, T.: A bottom-up pointer analysis using the update history. Information & Software Technology 51(4), 691–707 (2009)
15. Khedker, U.P., Sanyal, A., Karkare, A.: Heap reference analysis using access graphs. ACM TOPLAS 30(1) (2007)
16. Kuncak, V., Lam, P., Rinard, M.C.: Role analysis. In: Launchbury, J., Mitchell, J.C. (eds.) POPL, pp. 17–32. ACM (2002)
17. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: Cartwright, R. (ed.) PLDI, pp. 56–67. ACM (1993)
18. Larus, J.R., Hilfinger, P.N.: Detecting conflicts between structure accesses. In: Wexelblat, R.L. (ed.) PLDI, pp. 21–34. ACM (1988)
19. Matosevic, I., Abdelrahman, T.S.: Efficient bottom-up heap analysis for symbolic path-based data access summaries. In: Eidt, C., Holler, A.M., Srinivasan, U., Amarasinghe, S.P. (eds.) CGO, San Jose, CA, USA, pp. 252–263 (March 2012)
20. Murphy, B.R., Lam, M.S.: Program analysis with partial transfer functions. In: Lawall, J.L. (ed.) PEPM, pp. 94–103. ACM (2000)
21. Sălcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
22. Steensgaard, B.: Points-to analysis in almost linear time. In: Proc. 1996 ACM Symp. POPL, St. Petersburg, FL, USA, pp. 32–41. ACM Press (January1996)
23. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proc. CASCON 1999, pp. 125–135 (1999)
24. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Moshovos, A., Steffan, J.G., Hazelwood, K.M., Kaeli, D.R. (eds.) CGO, pp. 218–229. ACM (2010)