Path-Sensitive Data Flow Analysis Simplified

Kirsten Winter[1], Chenyi Zhang[1], Ian J. Hayes[1], Nathan Keynes[2], Cristina Cifuentes[3], Lian Li[3]

[1] School of ITEE, University of Queensland, Australia
[2] Oracle Brisbane, Australia
[3] Oracle Labs, Brisbane, Australia

Abstract. Path-sensitive data flow analysis pairs classical data flow analysis with an analysis of feasibility of paths to improve precision. In this paper we propose a framework for path-sensitive backward data flow analysis that is enhanced with an abstraction of the predicate domain. The abstraction is based on a three-valued logic. It follows the strategy that path predicates are simplified if possible (without calling an external predicate solver) and every predicate that could not be reduced to a simple predicate is abstracted to the *unknown* value, for which the feasibility is undecided. The implementation of the framework scales well and delivers promising results.

1 Introduction

Data flow analysis (DFA) [Kil73,NNH99] is a static analysis technique for compiler optimisation and program verification that scales to large code bases. In classical DFA, efficiency is achieved by conservatively over-approximating the behaviour of a program, taking all possible paths into account. When the framework is applied on static program analysis, the result is not precise: firstly, it loses information at join points of the program when data flow values from different paths are merged, and secondly, it may report bugs that arise from infeasible paths. This can lead either to a large number of false positives (i.e., a tool reports non-existing bugs) if the analysis reports all bugs that might occur on some paths, or a large number of false negatives (i.e., bugs that are missed by the analysis) if the analysis reports a bug only if it is encountered on all paths. On large code bases a high rate of reported false positives obstructs the debugging process whereas generally, a high rate of false negatives renders the analysis ineffective.

In order to detect most existing bugs with a low false positive rate, several approaches have been proposed to make DFA *path-sensitive.* Path-sensitive DFA collects path information which indicates feasibility or infeasibility of a path, and only reports bugs from feasible paths. Path information is given through the flow predicates that determine the flow of control in a program. During the analysis the flow predicates are combined to larger predicates, the satisfiability of which is in general undecidable if all path information is taken into account. Hence, full path sensitivity is hard to scale.

In this paper we propose a theoretical framework for a path-sensitive backward DFA which we enhance with an abstraction mechanism. The framework utilises Dijkstra's *weakest preconditions* and assertions added to the code to represent preconditions that are required for correctness. As such, assertions are a general means to indicate violations in the code and in our context they play the role of data flow facts. A feature of our approach is that path information and data flow facts are both encoded in the same predicate domain. Hence, they can be merged and the resulting predicate simplified. The abstraction is defined on the predicate domain to abstract from predicates that are too complex and to let the DFA procedure manipulate simple predicates only. We base this abstraction on a 3-valued logic which includes *unknown* as a third truth value. The abstraction maps each complex predicate onto a special predicate Δ, which is semantically *unknown* and hence can be either *true* or *false*.

The predicative backward DFA is implemented in the Parfait tool [CS08] which is a bug-checker built on top of LLVM [LA04]. We use the analysis to detect bugs such as memory leaks, use-after-free, double-free and free-of-non-allocated-pointer in sequential code. The results are encouraging and show that with the abstraction the analysis scales to code bases of over 6 million lines of code with a precision that delivers a false-positive rate of less than 5%.

The paper is organised as follows. Sections 2 and 3 recount the basic concepts of data flow analysis. The framework for a predicative backward DFA is introduced in Section 4 and its application is demonstrated in Section 5. Section 6 introduces our abstraction mechanism on the predicate domain and justifies the soundness for the approach. Section 7 reports on the experimental results when applying the implementation in the Parfait tool to the Solaris ON B20 source code. Section 8 discusses related work of path sensitive approaches in DFA. Section 9 concludes the paper.

2 Data flow graphs

We define a flowchart language that consists of states and a transition relation on states. The states and the transition relation are represented as a flow graph $G = (N, E, n_0, n_x)$ where N is a set of nodes, $E \subseteq N \times N$ a set of edges, $n_0 \in N$ a distinguished start node, and $n_x \in N$ a distinguished exit node. For an edge $e = (n, n') \in E$ we say n is the *source* of e, written as $src(e)$, and n' is the *destination* of e, written as $dst(e)$. A *path* π is a sequence of consecutive edges $e_1 e_2 \ldots$ satisfying $dst(e_i) = src(e_{i+1})$ for all i. The set of immediate predecessors of a node n is defined as $pred(n) = \{n' \mid (n', n) \in E\}$, and the set of immediate successors of a node n is $succ(n) = \{n' \mid (n, n') \in E\}$. A *program* is a tuple $Prog = (G, Var, effect)$, where G is the flow graph of the program, Var is the set of variables of the program, and edge labelling $effect : E \to \Phi$ where Φ is the set of statements (or their semantics) in the program. As usual, we define a program state as a mapping from variables to values.

For the analysis we enhance the program code with *assertions* (also called *assumptions* in the literature) which are specific to the analysis performed. They

```
void example(char *file){
  int err = 0, fd;
  int *tmp = malloc(..);
  if(tmp == Null) {
     err = 1;
     goto cleanup;
  }
  close(fd);
  free(tmp);
cleanup:
  if(err != 0) {
     free(tmp);
  }
}
```

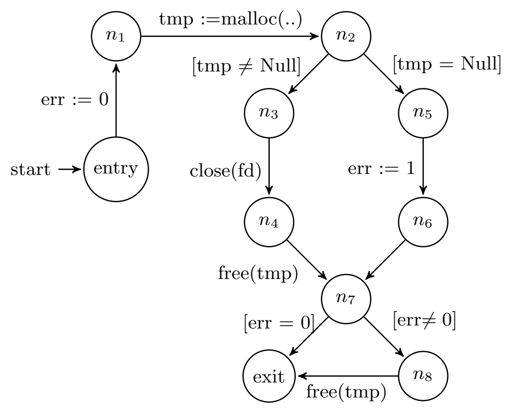

Fig. 1. An example program and its flow graph

are added to the code at particular points which are also specific to the problem to be analysed. We consider assertions as a type of statement in the program. Hence, the set of statements Φ includes assignment statements, guards (or flow conditions), and assertions.

Example 1. In Figure 1 we give an example flow graph on the right where the edges are labelled with the effects that correspond to the C code on the left. □

3 Data Flow Analysis

The classical data flow analysis (DFA) framework for *forward* and *backward* directed analysis, is defined as a tuple $\mathcal{L} = (\mathcal{D}, \sqcup, \sqsubseteq, \top, \bot, \mathtt{T}_b, \mathtt{T}_f)$, where we have $(\mathcal{D}, \sqcup, \sqsubseteq, \top, \bot)$ a complete semi-lattice with $\top \in \mathcal{D}$ the top element and $\bot \in \mathcal{D}$ the bottom element, $\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ a join operator, and $\sqsubseteq$ a partial order on $\mathcal{D}$ (with $x \sqsubseteq y$ iff $x \sqcup y = y$). Informally, $\mathcal{D}$ is a set of data flow values which abstractly represent the states of a program with respect to some specific characteristics. The set $\mathcal{D}$ together with an abstraction function α, which maps each concrete state onto an element in the abstract domain, constitutes the *abstract domain* (as defined in [CC77,RM07]). The function $\mathtt{T}_b : \Phi \to (\mathcal{D} \to \mathcal{D})$ is a *backward transfer function* that defines for each side effect the impact on the data flow values. Similarly, $\mathtt{T}_f : \Phi \to (\mathcal{D} \to \mathcal{D})$ is a *forward transfer function.*

Given a program *Prog* and a DFA framework $\mathcal{L}$, a data flow problem is to compute a mapping $D : N \to \mathcal{D}$ that assigns a data flow value to each node in the flow graph of *Prog* such that each node is labelled with the data flow value that is “achievable” at that point. To compute this mapping requires a fixpoint computation that is bound by the height of the semi-lattice in $\mathcal{L}$. For each node (simultaneously) the algorithm iteratively computes the following constraint:

(1) in the forward analysis and (2) in the backward analysis.

$$D_{i+1}(n) = \left(\bigsqcup_{n' \in \mathit{pred}(n)} \mathtt{T}_f(\mathit{eff}(n', n))(D_i(n')) \right) \tag{1}$$

$$D_{i+1}(n) = \left(\bigsqcup_{n' \in \mathit{succ}(n)} \mathtt{T}_b(\mathit{eff}(n, n'))(D_i(n')) \right) \tag{2}$$

for any $i \geq 0$, where $D_0(n) = \bot$ for all $n \in N$. This leads to a sequence of values $D_0(n) \sqsubseteq D_1(n) \sqsubseteq \ldots$, which terminates when a fixpoint is reached. As can be seen, the forward analysis requires the information from all predecessor nodes $n \in \mathit{pred}(n)$ and uses the function $\mathtt{T}_f$, whereas the backwards analysis builds on the information of the successor nodes, $n' \in \mathit{succ}(n)$, and uses $\mathtt{T}_b$.

The interpretation of the semi-lattice and the join operator $\sqcup$ depend on the analysis problem and the chosen approach. In the context of this paper, we target a so called *may* analysis which collects information that may be true on some paths and computes an *over-approximation* of the behaviour. The join operator $\sqcup$ is interpreted as set union $\cup$, and $\sqsubseteq$ is the subset relation $\subseteq$.

Example 2. As an example we use the DFA approach to solve the problem of memory leak detection of the local pointer variable `tmp` in the program of Figure 1. (Note that there might be other violations in the code that might be analysed at a later stage, e.g., an attempted closing of a file pointed to by `fd` which has not been opened). In a path-insensitive DFA (without information about the control flow) a violating path through nodes *entry*-n_1-n_2-n_5-n_6-n_7-*exit* will be reported as a potential memory leak. This path, however, is a false positive for two reasons: firstly, the path is infeasible due to the test on err, and secondly it also requires `tmp` to be `Null` (i.e., `malloc` to fail) in which case no memory is allocated and hence no leak has occurred. In this case, the information necessary to identify this report as a false positive is given through the flow condition $err = 0$ that is falsified by the preceding assignment $err := 1$. □

4 Predicative Backward DFA

In a backward DFA, the function D provides a conservative over-approximation of the program's state (in terms of its achievable data flow values) at each node. To achieve a more precise result and to rule out infeasible paths we are aiming at a *path-sensitive* analysis that collects control flow information along the paths through the flow graph. This leads us to a *predicative* backward DFA, in which $\mathcal{D}$ becomes the domain of predicates *Pred* and $D : N \to \mathit{Pred}$. *Pred* is the set of predicates that capture the states from which there exists a *feasible* path (usually as a conservative over-approximation) along which the data flow value, we aim to calculate, is achievable. We refer to *Pred* as the *predicate domain.*

The predicative data flow framework instantiates the simple DFA framework by choosing a particular abstract domain $\mathcal{D}$, transfer function $\mathtt{T}_b$ (in the backward case) and join operator $\sqcup$. In the following we provide detailed definitions of these constituents.

Predicate Domain. We define the predicate domain *Pred* as a set of predicates over program variables. We write $var(p)$ to denote the variables in predicate p. The set of predicates forms a lattice in which $p \sqsubseteq q$ is defined in terms of entailment: the logical implication of predicates that holds for all states, written $(p \Rrightarrow q)$. We let *true* be the top element of the lattice which is satisfied by all states, and *false* the bottom element, representing the empty set of states.

Transfer Function. The transfer function for the predicative backward DFA is given as the *predicate transformer* $\overline{wp} : \Phi \rightarrow (Pred \rightarrow Pred)$. We define this function based on the dual of Dijkstra's *weakest precondition* (wp) [Dij76,HFL01], namely $\overline{wp}(\mathit{eff})(p) = \neg wp(\mathit{eff})(\neg p)$. The dual of the weakest precondition, $\overline{wp}(\mathit{eff})(p)$, intuitively computes the set of pre-states from which there *exists* a possible execution of the statement *eff* such that p is satisfied after the statement. This leads us to the following rules:

$$\begin{array}{rcll}
\overline{wp}(\{A\})(p) & = & \mathit{def}(A) \Rightarrow \neg A \vee p & \text{(assertion)} \\
\overline{wp}([\,g\,])(p) & = & \mathit{def}(g) \Rightarrow g \wedge p & \text{(guard)} \\
\overline{wp}(v := E)(p) & = & \mathit{def}(E) \Rightarrow p[E/v] & \text{(assignment)} \\
\overline{wp}(S;R)(p) & = & \overline{wp}(S)(\overline{wp}(R)(p)) & \text{(sequential composition)} \\
\overline{wp}(S \sqcap R)(p) & = & \overline{wp}(S)(p) \vee \overline{wp}(R)(p) & \text{(non-deterministic choice)}
\end{array}$$

where $p[E/v]$ denotes the predicate in which all free occurrences of variable v are replaced by expression E, $(S;R)$ denotes the sequential composition of statements S and R, and $(S \sqcap R)$ denotes the non-deterministic choice between the two statements. With $\mathit{def}(A)$, $\mathit{def}(g)$ and $\mathit{def}(E)$ we denote the requirement that A, g and E, respectively, must be well-defined. In the following we assume the well-definedness of assertions, variables and expressions, which is subject to another analysis. An assertion is a condition that must be satisfied at a node, in the sense that condition $\neg A \vee p$ is used to tag paths that lead to states that *violate* A. Note that this transfer function allows us to collect path information (when conjoining a path predicate with the guard) as well as information of potential violations of assertions (when disjoining the path predicate with the negation of the encountered assertion).

Join Operator. Since the predicate transformer $\overline{wp}.\mathit{eff}$ provides the computation along one path, one computes the disjunction of predicates at join points which effectively models the non-deterministic choice of paths. Thus, we establish a simultaneous solution by computing for all $n \in N$ the weakest precondition $D(n)$ that satisfies the following data flow constraint in which the join operator becomes disjunction.

$$D_{i+1}(n) \;=\; \bigvee_{n' \in succ(n)} \overline{wp}(\mathit{eff}(n,n'))(D_i(n')).$$

Initialisation. We start the analysis with the bottom element and initialise the labelling of each node with *false*, i.e., $D_0(n) = \mathit{false}$ for all $n \in N$. Hence, for all $n \in N$, the fixpoint algorithm computes a sequence of values which are ordered by entailment, i.e, $D_0(n) \Rrightarrow D_1(n) \Rrightarrow \ldots$ A fixpoint is reached if $D_{i+1}(n) = D_i(n)$, for some $i \geq 0$. Note that a fixpoint may not be reachable within finite number of iterations in the presence of loops given the infinite domain *Pred*, nevertheless, we construct a prototype for the algorithm that quickly converges under the abstraction and simplification rules introduced in Section 6. The iterative application of the transfer function during the algorithm will result in a predicate at each node which represents the set of states that lead to erroneous paths ending at a final state where not all assertions are satisfied.

5 Applying the predicative backward DFA

We are using the predicative backward DFA as introduced above in the context of the static analysis tool Parfait [CS08] in which a potential bug list is established prior to the analysis. Each of these potential bugs is analysed in turn to decide if it is a real bug or a false positive. For each run of the analysis we *instrument* the analysis for the particular bug in focus. Instrumentation refers to formulating and automatically adding assertions to the code, as well as abstracting the statements that are specific to the potential bug. We demonstrate this using our example.

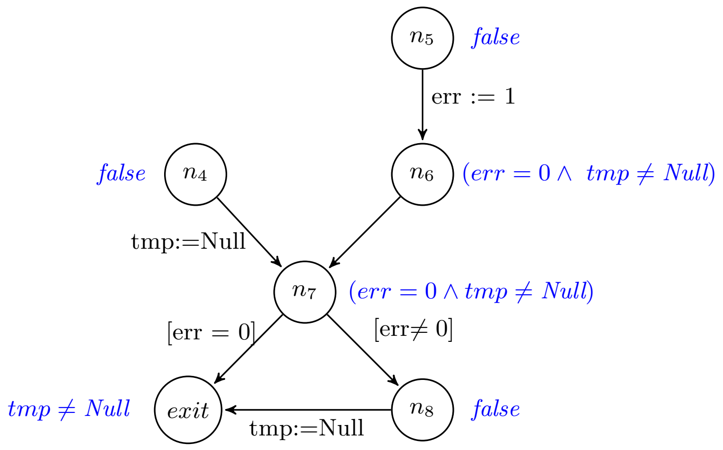

Fig. 2. Labelling the flow graph during memory-leak analysis

Example 3. We consider a potential memory leak in the code fragment introduced in Example 1 (see Figure 1). This example focuses on the pointer variable *tmp* which is allocated memory through the statement $tmp := malloc()$. To analyse this potential memory leak we assume that *malloc* has not failed (if it fails memory cannot leak). We abstract this statement by $tmp := NonNull$. Similarly, we abstract any occurrence of *free*(tmp) to $tmp := Null$ to capture deallocation. We replace the edge labelling in the flow graph in Figure 2 according to this abstraction.

The postcondition of this function is that the allocated memory has been freed at the end of the procedure. Using our abstraction, this is specified as $tmp = Null$. Since we want to find memory leaks, we label the exit node with the *inverse* of the postcondition specifying that memory has leaked, i.e., $tmp \neq Null$ (see Figure 2).

The predicative backward DFA labels each node n with a predicate $D(n)$ characterising the states at that node for which it is possible for a memory leak (i.e., a violation of the postcondition) to occur. A label of *false* at the entry node indicates that there are no paths leading to the exit where $tmp \neq Null$ is satisfied, and hence the code is free of memory leaks.

Initially, all nodes (except the exit node) are labelled with the predicate *false* and we start the analysis at the exit node. Applying $\overline{wp}$ along a backward traversal through the graph we label the nodes as indicated in Figure 2. At nodes n_4 and n_5 the analysis results in the predicate *false* which terminates the analysis (i.e., all nodes preceding n_4 and n_5 in Figure 1 are also labelled with *false*) as there are no more assertions added above those points. The result indicates that no memory leak is possible along any paths through the program. □

Another type of bug to be investigated is *free-of-non-allocated-pointer*. This can indicate a path on which *double-free* occurs or an attempted *free* after the allocation has failed. In this case we need to consider all possible outcomes of the memory allocation and hence a different kind of abstraction is required than for memory leak detection. We demonstrate the analysis in the following example.

Example 4. For an analysis to detect all paths along which a *free-of-non-allocated-pointer* occurs, we abstract the assignment $tmp := malloc()$ by the non-deterministic choice between the successful and the unsuccessful allocation of memory to the pointer tmp, i.e., $(tmp := NonNull) \sqcap (tmp := Null)$. The call *free*$(tmp)$ is abstracted to $\{tmp \neq Null\}; tmp := Null$. We modify the edge labelling in the flow graph to incorporate the abstraction. The assertion to be added to the code requires that memory space must be allocated to tmp in order to have a correct call to the *free* operation. We replace *free*(tmp) with an abstraction that includes the assertion $\{tmp \neq Null\}$ which must hold *before* the call to *free*(tmp). The assertions can be found in the sequential composition labelling the edges (n_4, n_7) and $(n_8, exit)$ in Figure 3.

Initially, all nodes (including the exit node) are labelled with *false*. Applying $\overline{wp}$ along a backwards traversal of the program graph in Figure 3 leads to a labelling of nodes as shown: Every node is labelled with a predicate characterising the states from which there is a possible violation. The resulting *true* labelling of the *entry* node indicates that there exists a path along which the assertion is violated and a *free* has been called on a non-allocated pointer.

Generally, a labelling $D(entry) \neq false$ indicates that there is a path that that starts with a state satisfying $D(entry)$ and leads to the bug in question through

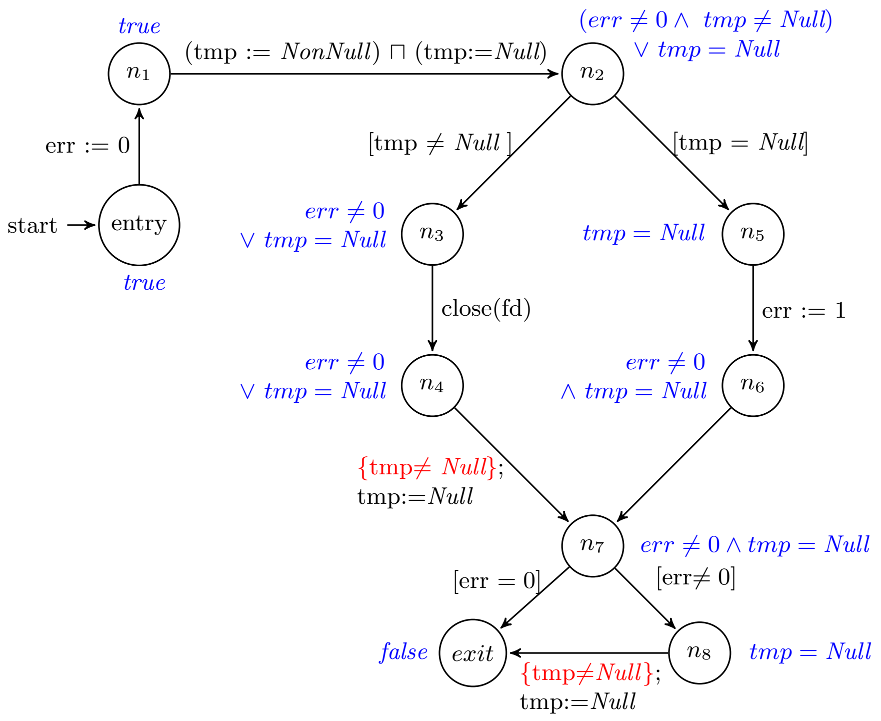

Fig. 3. Labelling the flow graph during the free-non-allocated-pointer analysis

the path *entry*-n_1-n_2-n_5-n_6-n_7-n_8-*exit*. In this particular case, we may further split the nondeterministic side effect of *malloc(..)* into $(tmp := NonNull)$ and $(tmp := Null)$, and apply the two parts of the transfer function on the predicates separately. This gives $\overline{wp}((tmp := NonNull))((err \neq 0 \wedge tmp \neq Null) \vee\ tmp = Null)$ which is the predicate $(err \neq 0)$, and $\overline{wp}((tmp := Null))((err \neq 0 \wedge tmp \neq Null) \vee\ tmp = Null)$ which is just *true*. As the predicate $(err \neq 0)$ is made *false* by the assignment edge $(entry, n_1)$, this reveals that it is the failed *malloc(..)* that has caused the *free-of-non-allocated-pointer* bug. □

As mentioned in Section 4, in the presence of loops, the iterative method may not converge to a fixpoint. In practice this can be handled by an abstraction that computes a less precise solution, which is introduced in the following section.

6 Simplifying the predicate expression

In practice the treatment of large predicates is costly in particular if an external predicate solver is invoked to simplify expressions. To avoid this complexity, we propose an abstraction that maps all predicates that cannot be easily simplified to a special predicate Δ representing *unknown*. Hence, the granularity of the abstraction is (inversely) related to the notion of simplification: the more predicates are simplified within our implementation, the less predicates will need to be abstracted.

As a first step we create an *embedding* of our 2-valued predicate domain *Pred* by defining a homomorphism h (i.e., h is injective and structure-preserving) that maps predicates into a 3-valued predicate domain, $Pred^3$. Apart from 2-valued predicates, $Pred^3$ also contains the symbol Δ. As a second step we define an abstraction function α from the 3-valued predicate domain into an abstract domain $Pred^A$ which contains predicates that are simplified into a disjunctive form (see Definition 3 in Section 6.2) as well as Δ. The two steps are depicted in Figure 4.

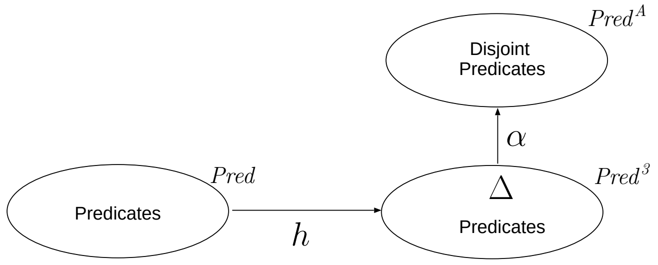

Fig. 4. Embedding and abstracting the predicate domain

Formulas over the embedding $Pred^3$ are captured by $\mathcal{L}^{\Delta}$, the *logic with unknown*, which is defined in Section 6.1. The definition of the abstraction α is given in Section 6.2.

6.1 The Logic with Unknown

Let P be the set of primitive predicates (e.g., relations representing flow conditions of a program). We define the syntax of the logic $\mathcal{L}^{\Delta}$ as follows:

$$\varphi := \mathit{true} \mid \mathit{false} \mid p \mid \Delta \mid \neg\phi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$$

with $p \in P$ and Δ as the special symbol for *unknown*. We use the 3-valued semantic domain $\mathcal{D} = \mathbb{P}_1(\{tt, \mathit{ff}\})$ (the powerset of Booleans excluding the empty set[4]) and the interpretation $\phi : \mathcal{L}^{\Delta} \to \mathcal{D}$, such that $\phi(\mathit{true}) = \{tt\}$, $\phi(\mathit{false}) = \{\mathit{ff}\}$, $\phi(p) \in \{\{tt\}, \{\mathit{ff}\}\}$ for all $p \in P$, and $\phi(\Delta) = \{tt, \mathit{ff}\}$ (either tt or ff, we don't know).

Remark 1. $\mathcal{L}^{\Delta}$ may be regarded as a restricted form of Kleene's three-value logic, it is different, however, in the following aspects.

– We have Δ in the syntax in addition to the three-value semantic domain.
– Every *known* primitive predicate p is always assigned with a definite meaning, i.e., a singleton in $\mathcal{D}$. Only the special symbol Δ remains *unknown*.

[4] The domain $\mathcal{D}' = \mathbb{P}(\{tt, \mathit{ff}\})$ includes the empty set $\{\}$ which would represent *contradiction*, the top element in Ginsberg's smallest non-trivial bilattice [Gin88].

– In Kleene's system there is no (nontrivial) tautology, but in our specialized logic $\mathcal{L}^{\Delta}$ we have valid formulas. For example $p \Rightarrow p$ is valid for any $p \in P$, simply because p cannot be unknown.

Semantics. Let $\oplus$ be a binary Boolean operator and $\ominus$ an unary Boolean operator. We lift these operators into $\mathcal{D}$ using the notation $[\![\oplus]\!] : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ and $[\![\ominus]\!] : \mathcal{D} \to \mathcal{D}$, respectively. The semantics is defined as point-wise application of the corresponding Boolean operator on the elements in the Boolean set, i.e., $\phi(\varphi_1 \oplus \varphi_2) = \{[\![\oplus]\!](b_1, b_2) \mid b_1 \in \phi(\varphi_1) \wedge b_2 \in \phi(\varphi_2)\}$ (the set of values $b_1 \oplus b_2$ for any b_1 and b_2 that are possible interpretations of φ_1 and φ_2, respectively), and $\phi(\ominus(\varphi)) = \{[\![\ominus]\!](b) \mid b \in \phi(\varphi)\}$ (the set of values $\ominus b$ for any b that is a possible interpretation of φ). In particular, for any $d \in \mathcal{D}$, $\phi(\mathit{true} \wedge d) = \phi(d)$ and $\phi(\mathit{false} \wedge d) = \{\mathit{ff}\}$. On Δ the lifted operators are resolved as follows

$$\begin{array}{lllllllll}
\phi(\Delta \wedge \mathit{true}) & = & \{\mathit{tt}, \mathit{ff}\}[\![\wedge]\!]\{\mathit{tt}\} & = & \{\mathit{tt} \wedge \mathit{tt}, \mathit{ff} \wedge \mathit{tt}\} & = & \{\mathit{tt}, \mathit{ff}\} & = & \phi(\Delta) \\
\phi(\Delta \wedge \mathit{false}) & = & \{\mathit{tt}, \mathit{ff}\}[\![\wedge]\!]\{\mathit{ff}\} & = & \{\mathit{tt} \wedge \mathit{ff}, \mathit{ff} \wedge \mathit{ff}\} & = & \{\mathit{ff}\} & = & \phi(\mathit{false}) \\
\phi(\Delta \vee \mathit{true}) & = & \{\mathit{tt}, \mathit{ff}\}[\![\vee]\!]\{\mathit{tt}\} & = & \{\mathit{tt} \vee \mathit{tt}, \mathit{ff} \vee \mathit{tt}\} & = & \{\mathit{tt}\} & = & \phi(\mathit{true}) \\
\phi(\Delta \vee \mathit{false}) & = & \{\mathit{tt}, \mathit{ff}\}[\![\vee]\!]\{\mathit{ff}\} & = & \{\mathit{tt} \vee \mathit{ff}, \mathit{ff} \vee \mathit{ff}\} & = & \{\mathit{tt}, \mathit{ff}\} & = & \phi(\Delta) \\
\phi(\neg\Delta) & = & [\![\neg]\!]\{\mathit{tt}, \mathit{ff}\} & = & \{\mathit{ff}, \mathit{tt}\} & = & \phi(\Delta) & &
\end{array}$$

and similarly, $\phi(\Delta \wedge \Delta) = \phi(\Delta \vee \Delta) = \phi(\Delta)$. With the standard predicate logic rules $((c \wedge d) \Leftrightarrow c) \equiv (c \Rightarrow d)$ and $((c \vee d) \Leftrightarrow d) \equiv (c \Rightarrow d)$, we can conclude from the above that $\mathit{false} \Rightarrow \Delta$ and $\Delta \Rightarrow \mathit{true}$, hence $\mathit{false} \Rightarrow \Delta \Rightarrow \mathit{true}$.

Information Order. Predicates are partially ordered by implication. We have $\mathit{false} \Rightarrow p \Rightarrow \mathit{true}$ for all predicates p, including Δ. However, Δ is incomparable to any other predicate in P in this ordering. To relate results in the embedding domain (and later on the abstract domain) with results in the concrete domain, we introduce an *information order* which lies orthogonal to the partial order on truth values (see also [Gin88,SRW02]).

Definition 1 (Information Order). *The* information order*, $\leq_i$, is a relation over formulas in $\mathcal{L}^{\Delta}$ such that for any formulas $\varphi_1, \varphi_2 \in \mathcal{L}^{\Delta}$, $\varphi_1 \leq_i \varphi_2 \;\Leftrightarrow\; \phi(\varphi_1) \supseteq \phi(\varphi_2)$.*

Intuitively, Δ provides less information to an analysis than any definite formulas, and from this definition it follows that $\Delta \leq_i \varphi$ for any formula $\varphi \in \mathcal{L}^{\Delta}$. As can be shown, the information order $\leq_i$ is a partial order over Pred^3 and all the logical operators are monotone with respect to $\leq_i$. In particular, for any predicates p and q, we have $\Delta \leq_i (\Delta \vee p) \leq_i q \vee p$ and $\Delta \leq_i (\Delta \wedge p) \leq_i q \wedge p$.

Simplification. Simplification is crucial for an effective path-sensitive DFA as it allows one to collapse predicates that otherwise grow very complex, but it is computationally expensive. Formulas in Pred^3 can be simplified using the rules of standard predicate logic. In our implementation some simplifications are realised, namely those that do not require expensive reasoning power. In particular, we restrict the simplification to *base predicates*, which commonly occur in programs where simple flags are used to control the flow.

Definition 2 (Base Predicate). *A* base predicate *is either true or false or a term of the form "x op c" where "x" is a variable, "c" is a constant value, and "op" is a binary operator from the set* $\{=, \neq, <, \leq, >, \geq\}$ *(tests) or* $\{\&_{all}, \&_{any}, \overline{\&}_{all}, \overline{\&}_{any}\}$ *(bit-field tests).*

The set of base predicates, denoted by *BasePred*, is a subset of the set of flow conditions, i.e, $BasePred \subseteq P$. From the definition it follows that for all $p \in BasePred$ the variable set $var(p)$ refers to at most one variable. Bit-field tests are defined based on C's *bit-wise and* operator "&" as outlined in Figure 5, where k is a constant bit mask.

$$\boxed{\begin{array}{l}(x\ \&_{all}\ k) \equiv (x\ \&\ k) == k\\ (x\ \&_{any}\ k) \equiv (x\ \&\ k) \neq 0\\ (x\ \overline{\&}_{all}\ k) \equiv (x\ \&\ k) == 0\\ (x\ \overline{\&}_{any}\ k) \equiv (x\ \&\ k) \neq k\end{array}}$$

Fig. 5: Bit field tests

We denote the simplification of formulas with the operator $\lceil\cdot\rceil : \mathcal{L}^{\Delta} \to \mathcal{L}^{\Delta}$. For any formula $\varphi \in \mathcal{L}^{\Delta}$ we have $[\![\varphi]\!] = [\![\lceil\varphi\rceil]\!]$, i.e., the simplification preserves the semantics. In particular, $\lceil\varphi \wedge true\rceil = \lceil\varphi\rceil$, $\lceil\varphi \vee true\rceil = true$, $\lceil\varphi \wedge false\rceil = false$, $\lceil\varphi \vee false\rceil = \lceil\varphi\rceil$, $\lceil\varphi \wedge \varphi\rceil = \lceil\varphi \vee \varphi\rceil = \lceil\varphi\rceil$.

For any predicate $a \neq \Delta$ we simplify $\lceil a \vee \neg a\rceil$ to *true* and $\lceil a \wedge \neg a\rceil$ to *false*. We exploit associativity, idempotence, and absorption of the Boolean operators to simplify predicates, e.g., $\lceil a \vee (a \wedge b)\rceil = a$. Bit-field tests can be simplified following the above, when taking into account the facts that $(x\ \&_{all}\ k) \equiv \neg(x\ \overline{\&}_{any}\ k)$ and $(x\ \&_{any}\ k) \equiv \neg(x\ \overline{\&}_{all}\ k)$.

To enable further simplification, we represent the right hand side of any $p \in BasePred$ that is a test (rather than a bit-field test) as a vector of integer intervals. That is, the predicate $x = k$ is represented as $x \in \langle[k, k]\rangle$ (i.e., $k \leq x \leq k$), and $x < k$ as $x \in \langle[Min, k-1]\rangle$, where *Min* is the smallest representable integer in the machine. This allows us to compute the disjunction and the conjunction of base predicates with the same left hand side, if they are of the interval type, by applying interval union and intersection, respectively. That is, $(x \in S) \vee (x \in T)$ simplifies to $x \in (S \cup T)$, and $(x \in S) \wedge (x \in T)$ simplifies to $x \in (S \cap T)$, where the operators $\cup$ and $\cap$ are defined as merging and intersecting, respectively, of ordered sequences of intervals. The empty interval $\langle[\,]\rangle$ is simplified to *false* and the maximal range $\langle[Min, Max]\rangle$ to *true*. Note, that any predicate whose right hand side is represented as an interval vector is considered a base predicate.

6.2 Abstraction

The intention of our abstraction scheme is to keep only simple predicates and map all other (complex) predicates onto Δ. By "simple" predicates we mean predicates in disjunctive normal form (DNF) that consist of clauses of base predicates and negated assertions only. We consider the negation of assertions since the transfer function $\overline{wp}$, when applied to an assertion A and postcondition p, constructs the disjunction of the *negated assertion* $\neg A$ and p (i.e., $\overline{wp}(\{A\})(p) = def(A) \Rightarrow \neg A \vee p$). Hence the negation of the assertion will appear in the abstract predicate domain. We call the predicates in the abstract predicate domain *disjoint predicates* and they are defined in terms of the set of base predicates, $BasePred$, and the set of negated assertions, $Assertion$.

Definition 3 (Disjoint Predicate). *The set of* disjoint predicates*, denoted as* DisjointPred*, comprises predicates of the form* $p = p_1 \vee \ldots \vee p_n$ *such that for all* $1 \leq i \leq n$*, either* $p_i \in BasePred \cup Assertion \cup \{\varDelta\}$ *or* $p_i = b \wedge a$ *with* $b \in BasePred$ *and* $a \in Assertion$*.*

That is, the abstraction maintains base predicates as well as assertions which take the role of data flow facts. For a particular analysis $Assertion$ is usually a singleton (e.g., in Example 3, $Assertion = \{tmp \neq Null\}$, and in Example 4, $Assertion = \{tmp = Null\}$). In that sense $Assertion$ is a parameter to the predicative DFA framework that can be instrumented for a variety of analysis problems.

We define the abstract domain $Pred^A$ as the set of disjoint predicates and the unknown symbol (as shown in Figure 4). The simplification rules are applied to the formulas whenever possible immediately after the transfer function is applied and before abstraction is performed. We specialise the DFA computation as follows, where $D^A(n)$ represents the predicative data flow value at node n in the abstract domain.

$$D^A_{i+1}(n) \ = \alpha\left(\left\lceil \bigvee_{n' \in succ(n)} \lceil \overline{wp}(\mathit{eff}(n,n'))(D^A_i(n')) \rceil \right\rceil\right) \tag{3}$$

Since the result is computed as a value in $Pred^A$, the size of a disjoint predicate is linearly bounded by the number of variables in the program. This is because a predicate in $Pred^A$ can have at most $|N|+1$ disjuncts (or clauses) each as a base predicate associated with a distinct program variable, or the unknown predicate $\varDelta$ as defined in the abstraction α outlined below.

A number of specialised simplification rules are introduced for disjoint predicates. These more complex simplifications also preserve the semantics of the predicate. For example, for merging predicates from the *true* and *false* branches at an if-then-else node, we apply $\lceil (p \wedge (\varphi \vee \varphi_1)) \vee (\neg p \wedge (\varphi \vee \varphi_2)) \rceil = \varphi \vee \lceil (p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2) \rceil$, i.e., we extract the common part out of the disjoint predicates from both branches before simplifying the conjunctions. The final result of simplification is a disjoint predicate in DNF, i.e., in the form of $\bigvee_{i \in I} \varphi_i$ such that each clause φ_i is a single base predicate or an assertion, or a conjunction of a base predicate and an assertion, or predicate $\varDelta$ (see Definition 3). The abstraction function, $\alpha : Pred^3 \rightarrow Pred^A$, is defined as follows.

- $\alpha(\varDelta) = \varDelta$.
- For each $p \in BasePred \cup Assertion$, we have $\alpha(p) = p$ and $\alpha(\neg p) = \neg p$.
- If φ is a conjunction of the form $a \wedge \varphi'$ with assertion a, we have $\alpha(\varphi) = a \wedge \alpha(\varphi')$. (Note that a clause in a disjoint predicate cannot have more than one assertion as they can only be introduced via disjunction.)
- If φ is a conjunction of more than one base predicate, we have $\alpha(\varphi) = \varDelta$. In particular, $\alpha(p \wedge q) = \varDelta$ if $var(p) \neq var(q)$, since $p \wedge q$ cannot be simplified into a single base predicate. Similarly, if $var(p) = var(q)$ and $p \wedge q$ cannot be simplified into a single base predicate, the abstraction results in $\varDelta$.

– An abstract disjoint predicate is given as the disjunction of the abstracted clauses, i.e., $\alpha(\bigvee_{i\in I}\varphi_i) = \bigvee_{i\in I}\alpha(\varphi_i)$.

It is obvious from this definition that α is monotone with respect to the information order, i.e., $\phi(\alpha(p)) \leq_i \phi(p))$ for all predicates p. Moreover, in the presence of loops the DFA on the abstract domain will in most cases converge to a simple predicate or to $\varDelta$. If a base predicate does not converge, e.g., when it extends its range in each iteration, the analysis exits the loop with an approximation after a small number of iterations.

Example 5. Applying the outlined abstraction scheme to the flow graph in Example 4 (see Figure 3) does not change the labelling. Hence we modify the example slightly and change the flow conditions: We label the edge (n_2, n_3) with $[fd \neq -1]$ and the edge (n_2, n_5) with $[fd = -1]$. This yields a more complex predicate labelling node n_2, namely $D(n_2) = ((err \neq 0 \vee tmp = \mathit{Null}) \wedge fd \neq -1) \vee (tmp = \mathit{Null} \wedge fd = -1)$ which reduces to $(tmp = \mathit{Null}) \vee (err \neq 0 \wedge fd \neq -1)$. The abstraction simplifies this predicate to $D(n_3)^A = (tmp = \mathit{Null}) \vee \varDelta$. (The labelling for nodes n_1 and *entry* remains unchanged, i.e., equals *true*.) The result indicates that if $(tmp = \mathit{Null})$ then there exists a path from n_3 that does lead to a *free-of-non-allocated-pointer*. Otherwise, a *free-of-non-allocated-pointer* might occur along some path from n_3, we do not know. □

From the abstraction we require that the results computed in the abstract domain Pred^A coincide with the results that are computed in the original domain *Pred*, unless $\varDelta$ is reported. If the analysis results in $\varDelta$, it is inconclusive and no statement can be made whether an assertion was violated or not. In that sense $\varDelta$ comprises both possibilities, *true* and *false*, as its semantic value suggests, namely $\phi(\varDelta) = \{tt, \mathit{ff}\}$. If the analysis results in the entry node being labelled with $p \vee \varDelta$ it indicates that there is a bug along the paths where p holds, on all other paths, there might be a bug, we do not know. Following this observation, soundness can be stated on the basis of the information order.

Theorem 1 (Soundness). *The predicative DFA on the abstract domain* Pred^A *is sound with respect to the information order* $\leq_i$*, i.e., for all nodes* $n \in N$*, we have* $D^A(n) \leq_i D(n)$*.*

The proof of this result follows straightforwardly from the monotonicity of the abstraction with respect to the information order and the fact that the simplification is semantic-preserving, i.e., $\lceil\varphi\rceil = \varphi$. We conclude that the final result $D^A(n)$ iteratively computed from Equation 3 is either a precise answer, or an *unknown*.

7 Experimental Results

The path-sensitive framework has been implemented in the static bug checking tool Parfait [CS08]. Parfait is currently used for the detection of memory leaks, use-after-free, double-free, free-of-non-allocated, and other bug-types. During the bug checking procedure, Parfait first populates a list of *potential bugs* generated by a path-insensitive DFA, and then applies the path-sensitive backwards DFA to verify whether the reported bugs in the list are real bugs (i.e., occurring along a feasible path). The form of disjoint predicates with simple disjuncts in the analysis is the key to scalability, which enables Parfait to spend a relatively short amount of time processing millions of lines of code. Nevertheless, this simple abstraction does not significantly impact the precision of the algorithm, as the abstract domain closely mirrors the way in which programmers typically manage control flow in practice, i.e., by setting up one or two constant flag variables that are later identified in the cleanup phase at the end of the program.

Parfait also supports inter-procedural analysis by summarising effects of the functions. We seed the process with predefined summaries for the common standard C library functions such as `malloc` and `free` and also generate function summaries for other functions. These are propagated bottom-up through the call graph. In the case of memory-leak analysis, function summaries would include information about allocation and de-allocation of pointer variables, stores into any pointer variables, and escapes of any pointer values (by storing into globally accessible memory or returning a pointer value). We then perform an intra-procedural data flow analysis over each function that contains a potential bug, making use of the summaries generated for each call site.

For each potential bug to be examined, the predicative DFA returns one of the four cases: a non-false value (definitely a bug), *false* (definitely not a bug), Δ (unknown), and $p \vee \Delta$ (definitely a bug along some paths). We observe, however, that if a potential bug is a false alarm (i.e., not reachable), then most of the time it is reported as *false*, thanks to the way in which most programmers handle control flow with simple constant flags. Therefore, in the current version we choose to report both the non-false and the Δ cases as (potential) bugs, which consequently enables the tool to report significantly more bugs, but also permits false positives at a tolerable low rate. The large-scale experiment data, which was manually evaluated, further justifies our decision.

The current (inter-procedural) version of Parfait has been run over 6 million lines of the Solaris ON B20 source code in X4270 server equipped with 2×3.3GHz Xeon X5680 CPUs and 144Gb RAM, running Solaris 11.1. Our experiment spreads eight threads in parallel and completes in 24 minutes, reporting 674 memory-leak bugs with a false positive rate of 4.6% (31/674).

8 Related Work

As one of the early works on path-sensitive data flow analysis Holley and Rosen [HR80] proposed a general approach to improve the precision by computing path feasibility under a given set of assertions on variable values, and construct a new problem that contains only qualified paths. Bodík et al. in [BGS97] developed a strategy to detect infeasible paths using the notion of *branch correlations* (i.e., the dependence of a branch predicate on a previous branch or program statement). The process is demand-driven: at branch b with predicate

p a query about the satisfiability of p is raised which is propagated backwards in the flow graph until it is resolved along all paths. The resolution of the query is then passed forward to branch b that caused the query and the flow graph labelled accordingly. Fisher et al. [FJM05] base their analysis on a structure called a *predicated lattice* whose elements are mappings from predicates to dataflow values. In this framework dataflow values are only merged in the case when they are associated to the same predicate. Their analysis works in a forward fashion and hence requires a set of predicates P as input. If P is not sufficient to produce a precise result they iteratively refine P and add more predicates according to the false positives encountered (similar to lazy abstraction [HJMS02]). This process is repeated until the analysis provides a precise (or precise enough) solution.

Property simulation has been proposed by Das et al. [DLS02,HYD05,DDY06] and implemented in the ESP tool. The analysis is also performed in a forward fashion. The framework is based on an adjustable abstraction which represents symbolic states of the program. A property state machine specifies the property pattern to be checked and is exercised in parallel with the program code (through instrumentation of the program), where a simulation state represents path information. The tool applies a decision procedure to reason about feasibility of the paths.

Rival and Mauborgne [RM07] build a general framework for *trace partitioning* which allows for path sensitivity. The idea is to extend the general data flow analysis (following the abstract interpretation framework of [CC77]) with *tokens* which allow the partitioning of the state space, qualifying classes of states according to the history of execution. The set of tokens T is variable in the framework and can be set according to the analysed problem. If T is a singleton then the analysis coincides with the classical DFA.

All existing works in the literature for path sensitive program analysis separate the analysis on paths from the analysis on data, in a way that first explores the correlated predicates or properties to settle feasible path flows, and subsequently applies the analysis only on the feasible paths. Our work integrates data flow analysis with predicate analysis in a general weakest precondition framework. This allows us to merge and simplify paths predicates and data flow values (which are predicates here) during the analysis for better scalability.

9 Conclusion and Outlook to Future Work

This paper proposed a path-sensitive data flow analysis framework which combines path predicates with data flow values. The transfer function is based on Dijkstra's weakest precondition to reason about data flow values or violations via assertions. This approach supports the analysis of a large variety of bug patterns, such as locking-not-followed-by-unlocking and use-after-free, which can be simply described as a relationship between two distinct program points along a feasible path. We also introduced effective simplification and abstraction to facilitate the efficiency of the analysis. The work has been implemented in the static bug checker Parfait and the result scales to programs with millions of lines of code.

As further experiments, we are interested in refining the abstraction to include more complex predicates (e.g., allow for disjoint predicates with clauses with more than two conjuncts). This would enable us to relate the trade-off between efficiency and preciseness to the granularity of the abstraction.

In comparison to flow-sensitive forward analysis, predicative backward DFA has the advantage of being able to discover relevant predicates at an early stage (i.e., by selecting predicates that guard bug related assertions). However as many DFA problems can only be encoded in a forward analysis, it is of interest to explore forward DFA in the future, taking advantage of strongest postconditions and effective ways to extract predicates related to the problem.

References

[BGS97] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proc. of ESEC/FSE*, pages 361–377. ACM, 1997.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, pages 238–252. ACM, 1977.

[CS08] C. Cifuentes and B. Scholz. Parfait – designing a scalable bug checker. In *Proc. of the Static Analysis Workshop*, pages 4–11. ACM, 2008.

[DDY06] D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In *Proc. of SAS, LNCS 4134*, pages 425–442. Springer, 2006.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of PLDI*, pages 57–68. ACM, 2002.

[FJM05] J. Fisher, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *Proc. of ESEC/FSE*, pages 227–236. ACM, 2005.

[Gin88] M. Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[HFL01] I. J. Hayes, C. J. Fidge, and K. Lermer. Semantic characterisation of dead control-flow paths. *IEE Proceedings—Software*, 148(6):175–186, 2001.

[HJMS02] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL*, pages 58–70. ACM, 2002.

[HR80] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. In *Proc. of POPL*, pages 68–82. ACM, 1980.

[HYD05] H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. In *Proc. of PASTE*, pages 52–58. ACM, 2005.

[Kil73] G. A. Kildall. A unified approach to global program optimization. In *Proc. of POPL*, pages 194–206. ACM, 1973.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, 2004.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 1999.

[RM07] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM TOPLAS*, 29, August 2007.

[SRW02] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.