The Undefined Domain: Precise Relational Information for Entities that Do Not Exist[1]

Holger Siegel, Bogdan Mihaila, and Axel Simon

Technische Universität München, Institut für Informatik II, Garching, Germany
firstname.lastname@in.tum.de

Abstract. Verification by static analysis often hinges on the inference of relational numeric information. In real-world programs, the set of active variables is often not fixed for a given program point due to, for instance, heap-allocated cells or recursive function calls. For these program points, an invariant has to summarize values for traces E where a variable x exists and values for traces N where x does not exist. Non-relational domains solve this problem by copying all information on x in traces E to those in N. Relational domains face the challenge that the relations in traces E between x and other variables cannot simply be replicated for the traces N. This work illustrates this problem and proposes a general solution in form of a co-fibered abstract domain that forwards each domain operation to operations on a child domain. By tracking which variables are undefined, it transparently stores suitable values in the child domain thus minimizing the loss of relational information. We present applications in heap abstractions and function summaries.

1 Introduction

Static analyses that are based on relational numeric domains are often restricted to programs with limited dynamic memory allocation and without recursive functions [2]. Analyses that explicitly target heap manipulating programs often represent their state using a logic formula, e.g. separation logic [7] or three-valued logic analysis (TVLA) [8]. Combining them with relational numeric domains is not straightforward. In particular, problems occur when the numeric domain has to track a changing number of memory cells or when it has to deal with uninitialized variables. The following examples illustrate these problems.

Non-existing memory regions. In the C program in Fig. 1, a region x is allocated in one conditional branch but not in the other. When this program is analyzed using some abstract domain, the resulting abstract state of the then-branch has to be merged with that of the else-branch. The desired invariant is that x is initialized to 3 if p points to x. When using the Intervals domain, at least the fact that $x = 3$ can be inferred by allowing x to be mapped to an explicit empty interval $\bot$. Specifically, by using this bottom value $\bot$ as the value of x in the else-branch as shown in the first row of the table, the join of the branches retains

[1] This work was supported by DFG Emmy Noether programme SI 1579/1.

```
if (rnd()) {
  p = malloc(4);
  *p = 3;
} else {
  p = NULL;
}
```

Intervals	$\{p = 1 \cdot \&x,\ x = 3\} \sqcup \{p = 0 \cdot \&x,\ x = \bot\}$ $= \{p \in \{0, 1\} \cdot \&x,\ x = 3\}$
Polyhedra	$\{p = 1 \cdot \&x,\ x = 3\} \sqcup \{p = 0 \cdot \&x,\ x = \top\}$ $= \{p \in \{0, 1\} \cdot \&x,\ x = \top\}$
Polyhedra as Undefined child	$\{p = 1 \cdot \&x, x = 3, f_x = 1\}$ $\sqcup \{p = 0 \cdot \&x, x = 3, f_x = 0\}$ $= \{p = f_x \cdot \&x, x = 3, f_x \in \{0, 1\}\}$

Fig. 1. Non-existing regions.

```
if (rnd()) {
  x = 1;
  y = 2;
} else {
  x = 0;
}
```

Polyhedra/ Intervals	$\{x = 1, y = 2\} \sqcup \{x = 0, y = \top\}$ $= \{x \in \{0, 1\}, y = \top\}$
Polyhedra as Undefined child	$\{x = 1, f_x = 1, y = 2, f_y = 1\}$ $\sqcup \{x = 0, f_x = 1, y = 2, f_y = 0\}$ $= \{x \in \{0, 1\}, f_x = 1, y = 2, f_y = x\}$

Fig. 2. Non-initialized variables.

the information that x can only contain value 3. However, relational numeric domains typically model *smash products*, i.e. they do not allow assigning the "empty" value $\bot$ to individual variables since this always implies an empty set of relations and, thus, a domain state that is $\bot$. Thus, in contrast to non-relational domains, x must map to a value. Consider choosing $x = \top$. In this case the second table row shows that the relational Polyhedra domain [4] loses the information that $x = 3$ in the else-branch. In order to retain as much information as possible, we propose a copy-and-paste operation that can be used to add missing variables with values that are more precise than $\top$, so that $x = 3$ is retained in the join.

Non-initialized variables. Now we consider a precision loss that occurs in the C program in Fig. 2. Here, one conditional branch initializes variables x and y, whereas the other branch only initializes variable x, leaving y undefined. When the resulting states are joined, y has to be introduced in the latter state with an unrestricted value $\top$, giving the joined state $\{x = 1, y = 2\} \cup \{x = 0, y = \top\}$. However, introducing variables with value $\top$ can lead to loss of precision. In particular, the implication $x = 1 \Rightarrow y = 2$ is lost in domains whose state is a convex set. For instance, when using the relational Polyhedra domain, the joined state $\{x \in \{0, 1\}, y = \top\}$ (shown in the first row of the table) is only as precise as the join over the intervals, in that any relation between x and y is lost.

As a solution to these two problems of non-existing regions and of non-initialized variables, we propose a dedicated abstract domain called the *Undefined* domain which is parameterized over another numeric domain, that we call the *child domain*. The child can be an arbitrary numeric domain. We require that non-initialized *and* non-existent variables are introduced as $\top$. The Undefined domain then transparently inserts placeholder values using a so-called copy-and-paste operation. It additionally tracks a flag f_x that indicates if vari-

able x is defined, thereby enabling the child domain to infer relations with this flag, e.g. "x is defined iff p points to x". In order to introduce the Undefined domain, we now exemplify how it can be employed to analyze the examples.

We illustrate the Undefined domain by performing an abstract interpretation of the program in Fig. 1. The resulting state of the then-branch is represented by the child state $s_t = \{p = \&x, x = 3, f_x = 1\}$, where x is the content of region x and f_x is a flag that indicates whether x is defined or not. Since flag f_x has value 1, variable x is defined and it has value 3 stored in the child domain. Furthermore, the state in the else-branch $\{p = 0\}$ does not contain the variable x, so that it has to be added with $x = \top$. The Undefined domain observes this undefined value of x and replaces it by a copy of x from s_t, yielding the child state $s_e = \{p = 0, x = 3, f_x = 0\}$. Note here that the Undefined domain has added $f_x = 0$, indicating that x is not defined, and the value of x that is stored in the child domain has to be ignored. As shown in the second row of the table, joining both states results in the child state $\{p = f_x \cdot \&x, x = 3, f_x \in \{0, 1\}\}$, where p can only point to x if f_x has value 1, which means that x is defined. The Undefined domain has retained the information that region x can only contain value 3, although the undefined value of x is modeled by $\top$ instead of $\bot$.

Now consider analyzing the program in Fig. 2 using the Undefined domain with the Polyhedra domain as its child. The resulting state of the then-branch is represented by the child state $\{x = 1, f_x = 1, y = 2, f_y = 1\}$. Here, flags f_x and f_y have value 1, indicating that x and y are defined. The resulting state of the else-branch is modeled by the child state $\{x = 0, f_x = 1, y = 2, f_y = 0\}$. Flag f_y has value 0, indicating that y appears to have the value $\top$ at the interface of the Undefined domain. As before, the Undefined domain has used the value of y from the then-branch as a placeholder value. As shown in the second row of the table, the joined child state $\{x \in \{0, 1\}, f_x = 1, y = 2, f_y = x\}$ now indicates that $x = 1$ implies $f_y = 1$ and thus $y = 2$, an invariant that is retained although the state is approximated by the Polyhedra domain.

As shown in the examples, existing numeric domains can be wrapped by the Undefined domain. The resulting domain is a drop-in replacement for the original numeric domain. The Undefined domain transparently manages flags for all variables that may be undefined, thereby ensuring that all operations on the domain are sound even if some of the variables mentioned in the operations are undefined. We provide an implementation of the Undefined domain that partitions the flags into groups of flags with equal valuations. By collapsing each group into one single flag, it minimizes the required number of flag variables. In summary, this paper makes the following contributions:

- We describe how existing numeric domains can be enabled to incorporate program states of different size in one abstract state.
- We define the *Undefined* abstract domain that translates domain operations to operations on a child domain such that relational information can be inferred in these situations.
- We illustrate the precision of our approach by presenting examples that perform dynamic heap allocation and summarize calls to procedures.

After providing a formal basis, Sect. 3 defines the Undefined domain. Its utility for common analysis tasks is shown in Sect. 4 before Sect. 5 presents our experimental evaluation. Section 6 discusses related work and Sect. 7 concludes.

2 The *Undefined* Domain

Numeric domains may provide operations that change the *support set* of a numeric state, that is, the set of variables for which the domain holds numeric valuations. Joining and comparing states with different support sets is often preceeded by a process that makes their support sets equal. We follow [11] and describe domains with non-fixed support sets as *co-fibered domains*. This construction allows to systematically derive variants of the compare and join operations that adjust the support sets themselves. We first give a definition of numeric domains before we introduce the *Undefined* domain as an abstract numeric domain.

Definition 1 (Numeric Domain). *A* numeric domain *is given by a tuple* $(D, \sqsubseteq_D, \sqcup_D, T_D)$ *where* D *is the set of* states, $\sqsubseteq_D$ *is a pre-order,* $\sqcup_D$ *is a binary function, such that* $s \sqsubseteq_D s \sqcup_D t$ *and* $t \sqsubseteq_D s \sqcup_D t$ *for all* $s, t \in D$*, and* $T \subseteq D^D$ *is a set of monotonic* transfer functions*.*

Let $\mathbb{X}$ be the set of program variables. In this work, we assume that each numeric state $s \in D$ has a *support set* $\chi(s) \subseteq \mathbb{X}$ that represents the set of variables for which state s holds valuations. Then each state $s \in D$ represents a set of vectors of dimension $|\chi(s)|$. Since program variables may be introduced and removed during a program run, the numeric domain must provide operations that add or remove variables to and from the support set. Removing a variable x from a state $s \in D$ with $x \in \chi(s)$ is denoted by a function $drop_{D,x} : D \to D$. Adding an unrestricted variable x to a state $s \in D$ with $x \notin \chi(s)$ is denoted by a function $add_{D,x} : D \to D$. These functions are lifted to sets of variables by repeated application of *add* and *drop* operations, that is, $add_{D,X} := \bigcirc_{x \in X} add_{D,x}$ and $drop_{D,X} := \bigcirc_{x \in X} drop_{D,x}$ with $\bigcirc$ denoting function composition.

Comparing and joining two states $s \in D$ and $t \in D$ with different support sets requires to add missing variables to s and t beforehand. Following [11], we capture this behaviour by requiring that the pre-ordered set $(D, \sqsubseteq_D)$ together with the morphisms $add_{D,X}$, $drop_{D,X}$ for $X \subseteq \mathbb{X}$ forms a co-fibered domain. Specifically, we require the following equivalence:

$$s \sqsubseteq_D t \Leftrightarrow add_{D,\chi(t)\setminus\chi(s)}(s) \sqsubseteq_D add_{D,\chi(s)\setminus\chi(t)}(t).$$

For the sake of a generic presentation, we assume that all other transfer functions are divided into assignments $[\![y := f(x_1, \ldots, x_n)]\!]_D$ and tests $[\![f(x_1, \ldots, x_n) \leq 0]\!]_D$ with program variables $x_1, \ldots, x_n, y \in \mathbb{X}$ and an n-ary function f.

Example 1 (Concrete Domain). The *concrete domain* $(C, \sqsubseteq_C, \sqcup_C, T_C)$ models exact sets of vectors over $\mathbb{Z}$, that is, $C = \bigcup_{n \in \mathbb{N}} \wp(\mathbb{Z}^n)$. The join $s \sqcup_C t$ of two states s and t with $\chi(s) = \chi(t)$ is just the set union $s \cup t$. Analogously, for $\chi(s) = \chi(t)$ the comparison $s \sqsubseteq_C t$ is just $s \subseteq t$.

An abstract numeric domain over-approximates the state of another domain. Its semantics is determined by a *concretization function* γ that takes an abstract state to a concrete state.

Definition 2 (Abstract Domain). *A domain* $(A, \sqsubseteq_A, \sqcup_A, T_A)$ abstracts *a domain* $(D, \sqsubseteq_D, \sqcup_D, T_D)$ *if there is a* concretization *function* $\gamma : A \to D$*, such that* $s \sqsubseteq_A t \Rightarrow \gamma(s) \sqsubseteq_D \gamma(t)$ *and* $\gamma(s) \sqcup_D \gamma(t) \sqsubseteq_D \gamma(s \sqcup_A t)$ *for every* $s, t \in A$ *and for every* $\tau_D \in T_D$ *exists a* $\tau_A \in T_A$*, such that* $\tau_D \circ \gamma \sqsubseteq_D \gamma \circ \tau_A$*.*

This definition ensures that every domain operation of the abstract domain over-approximates the corresponding operation of the concrete domain.

Example 2. The previously mentioned domain of closed convex *Polyhedra* [4] abstracts the concrete domain C by over-approximating a set of vectors by the topological closure of their convex hull.

In Sect. 4 we will detail how the Undefined domain can improve the precision of the Polyhedra domain. In the remainder of this section, we assume an abstract domain $(A, \sqsubseteq_A, \sqcup_A, T_A)$ without making further assumptions about this domain.

2.1 The Undefined Domain

The *Undefined* domain is a functor domain [3]: Each state holds a state of a *child domain*, and domain operations are forwarded to domain operations on this child domain. Here, for each variable x of the Undefined domain, its child domain holds a variable x and a flag f_x. When $f_x = 1$ in the child domain, the value of x is given by the value of x in the child domain. When $f_x = 0$, variable x is unrestricted and the value stored for x in the child domain is just a placeholder. As a consequence, every numeric state of dimension n is modelled by a child state of dimension $2n$. We later detail how fewer dimensions suffice. We denote an Undefined domain that transforms a child domain $(D, \sqsubseteq_D, \sqcup_D, T_D)$ by $(U(D), \sqsubseteq_{U(D)}, \sqcup_{U(D)}, T_{U(D)})$. An element of the Undefined domain that has a child state $s \in D$ is denoted by $u \rhd s$ where u denotes the mapping from each x to its flag f_x. We fix the semantics of $U(D)$ by defining functions $\tilde{\gamma}_D$ that relate states of $U(D)$ to states of D.

Definition 3. *For every domain* $(D, \sqsubseteq_D, \sqcup_D, T_D)$*, function* $\tilde{\gamma}_D : U(D) \to D$ *is given by* $\tilde{\gamma}_D(u \rhd s) := (\bigcirc_{x \in \chi(s)} \tilde{\gamma}_{D,x})(s)$ *where* $\tilde{\gamma}_{D,x} : D \to D$ *is defined by* $\tilde{\gamma}_{D,x}(s) := drop_{D,f_x}([\![f_x = 1]\!]s \sqcup_D (add_{D,x} \circ drop_{D,x})([\![f_x = 0]\!]s))$ *for each* $x \in \mathbb{X}$*.*

Adding and Removing Dimensions. Removing a variable x from a state $u \rhd s \in U(D)$ consists of straightforwardly removing variable x and the corresponding flag f_x from the child state. Thus, we define $drop_{U(D),x}(u \rhd s) := u \rhd drop_{D,\{x,f_x\}}(s)$. Adding a variable x to a state $u \rhd s \in U(D)$ can be done in two different ways: One way is to simply add an unrestricted variable x and the corresponding flag f_x with value one to the child state s. Another way is to

add x with an arbitrary value and flag f_x with value zero to the child domain, indicating that x is undefined and the value held for x in the child domain has to be ignored. However, for abstract domains the chosen placeholder value of x can influence the precision of subsequent domain operations, namely join operations. In order to characterize the functions that may be used to introduce placeholder values for x, we first introduce the notion of X-Adders.

Definition 4. *A function $\varphi : D \to D$ is an* X-Adder *iff $\chi(\varphi(d)) = \chi(d) \cup X$ and $drop_{D,X}(\varphi(d)) = d$ for all sets of variables $X \subseteq \mathbb{X}$ and states $d \in D$ with $\chi(d) \cap X = \emptyset$.*

The intuition of an X-Adder φ is that it extends a state with new dimensions X that are bound to placeholder values. Given an X-Adder φ for domain D and variable set $x_1, \ldots, x_n$, we define operation $add^{\varphi}_{U(D),\{x_1,\ldots,x_n\}}(u \rhd s) := u \rhd [\![f_{x_1} = \cdots = f_{x_n} = 0]\!]\varphi(add_{D,f_{x_1},\ldots,f_{x_n}}(s))$. It remains to show that every operation $add_{D,X}$ on domain D has a corresponding operation on domain $U(D)$. Indeed, for every domain D and variable set X function $add_{D,X}$ itself is an X-Adder, and so $add^{add_{D,X}}_{U(D),X}$ corresponds to $add_{D,X}$.

Joining, Widening and Comparing States. Two states $u \rhd s$ and $u \rhd t$ with equal support sets $\chi(s) = \chi(t)$ are compared, joined or widened by simply performing these operations on their child states s and t. For states $u \rhd s$ and $u \rhd t$ with different support sets $\chi(s) \neq \chi(t)$, their support sets are made equal by performing add^{φ} operations on s and t before they can be compared, joined or widened. This allows for a degree of freedom, since an arbitrary X-Adder φ can be chosen for each add^{φ} operation. In the next section we will show how the precision of the Undefined domain can be improved by introducing an X-Addder that retains relational information between the variables X.

It is worth noting that the ordering given by this comparison operation slightly deviates from the pre-order obtained from constructing the co-fibered domain. There, the relation $u \rhd s \sqsubseteq u \rhd t$ holds for two states $u \rhd s$ and $u \rhd t$ with different support sets whenever two X-Adders φ and ψ *exist*, such that $add^{\varphi}_{D,\chi(t)\setminus\chi(s)}(s) \sqsubseteq_D add^{\psi}_{D,\chi(s)\setminus\chi(t)}(t)$, whereas the comparison operation only detects subset relations that can be established by a *previously chosen* pair of X-Adders. In fixpoint computations, this may lead to extra iterations, although termination is still guaranteed by widening. Consider, for example, the child states $s := \{x = 1, y = 1, f_y = 0\}$ and $t := \{x = 1\}$. Clearly, s and t describe the same state, but the comparison operation might obscure this by adjusting state t to $\{x = 1, y = 2, f_y = 0\}$, thereby necessitating one more fixpoint iteration.

Transfer Functions. An assignment $y := f(x_1, \ldots, x_n)$ is directly executed on the child domain. Since the resulting value y is only valid if all variables $x_1, \ldots, x_n$ are defined (that is, if all $f_i = 1$), the flag f_y is set to the conjunction $\bigwedge_{i=1}^{n} f_{x_i}$. A test $f(x_1, \ldots, x_n) \leq 0$ is performed by first splitting the state into one state where all $f_i = 1$ and one state where $f_i = 0$ for some i. The test is then performed on the former state, while the latter state remains unchanged. After that, both states are joined. Figure 3 shows the transfer functions for tests and assignments.

$$[\![y := f(x_1, \ldots, x_n)]\!]_{U(D)} u \rhd s$$
$$:= u \rhd [\![y := f(x_1, \ldots, x_n);\ f_y := \bigwedge_{i=1}^{n} f_{x_i}]\!]_D s$$
$$[\![f(x_1, \ldots, x_n) \leq 0]\!]_{U(D)} u \rhd s$$
$$:= u \rhd [\![f(x_1, \ldots, x_n) \leq 0; \bigwedge_{i=1}^{n} f_{x_i} = 1]\!]_D s \sqcup_D [\![\bigvee_{i=1}^{n} f_{x_i} = 0]\!]_D s$$

Fig. 3. Transfer functions for unary operations.

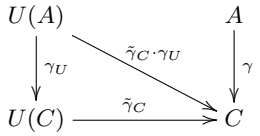

Fig. 4. Abstract domains and their concretizations.

2.2 Correctness of the Undefined Domain

In order to verify that domain $U(A)$ is indeed an abstraction of domain C, we first show that $U(A)$ is an abstraction of domain $U(C)$, and then we show that $U(C)$ is an abstraction of C. As sketched in Fig. 4, it follows immediately that domain $U(A)$ is also an abstraction of domain C. In the first step, we observe that domain $U(A)$ is an abstraction of $U(C)$.

Lemma 1. *If domain A abstracts domain C, then $U(A)$ abstracts $U(C)$.*

Proof. Given an concretization function $\gamma : A \to C$, a concretization function $\gamma_U : U(A) \to U(C)$ is given by $\gamma_U(u \rhd s) := u \rhd \gamma(s)$.

In the second step, we observe that the domain $U(C)$ is an abstraction of C.

Lemma 2. *Domain $U(C)$ abstracts domain C.*

Proof. For every Domain $(D, \sqsubseteq_D, \sqcup_D, T_D)$ and every $x \in \mathbb{X}$, function $\tilde{\gamma}_{Dx}$ is a concretization function. In particular, every function $add^{\varphi}_{U(D),x}$ abstracts function $add_{D,x}$. As a composition of such functions, function $\tilde{\gamma}_C$ also is.

The following theorem states the desired property that domain $U(A)$ is an abstraction of the concrete domain C: all operations on domain $U(A)$ over-aproximate the corresponding operations on the concrete domain C.

Theorem 1. *Domain $U(A)$ abstracts domain C.*

Proof. Since γ_U and $\tilde{\gamma}_C$ are concretization functions by Lemma 1 and 2, their composition $\tilde{\gamma}_C \circ \gamma_U : U(A) \to C$ also is.

The given semantics of the Undefined domain is still impractical, as it stores one additional flag variable for each variable in the child state, and it is incomplete, as it does not fully specify how missing variables are added. The next section describes how the number of flag variables can be reduced and suggests an X-Adder *copyAndPaste* that adds missing variables in a clever way: it copies relations between those variables that are missing in the respective other domain.

3 Practical Implementation of the *Undefined* Domain

In this section we propose an implementation of the Undefined domain that is practical in following two senses: firstly, it associates a flag with a set of variables rather than with each variable, thus yielding a scalable domain; secondly, it uses a copy-and-paste operation that transfers the valuations of whole sets of variables to another domain, thereby allowing for retaining relational information between variables of a partition. After some definitions, we consider each aspect in turn.

3.1 Definition of the Domain

Let $\mathcal{X} \subseteq \mathbb{X}$ denote the program variables and $\mathcal{F} \subseteq \mathbb{X} \setminus \mathcal{X}$ denote the variables used as flags. A state of the undefined domain $U(A)$ is given by $u \rhd a$ with child state $a \in A$ and a partial mapping $u : \mathcal{X} \dashrightarrow \mathcal{F}$. This mapping takes each variable in the state's support set to a flag that tracks whether this variable is defined. Thus, the support set of child state a is $\chi(a) = \mathrm{dom}(u) \cup \mathrm{img}(u)$ where $\mathrm{dom}(u)$ denotes the domain of u and $\mathrm{img}(u)$ denotes the image of u. We allow several program variables to map to the same flag variable, thereby inducing a partitioning of program variables. For each mapping u this partitioning is given by $\Pi(u) := \{u^{-1}(f) \mid f \in \mathrm{img}(u)\})$, where $u^{-1} : \mathcal{F} \to \wp(\mathcal{X})$ is the reverse relation of u. For better legibility, we sometimes denote u by its reverse relation. Thus, for $u = [x_0 \mapsto f_0, x_1 \mapsto f_1, x_2 \mapsto f_0, x_3 \mapsto f_1]$ we write $[f_0 \mapsto \{x_0, x_2\}$, $f_1 \mapsto \{x_1, x_3\}]$. We now detail how to manage flags when partitions change.

3.2 Making Partitions Compatible

Whenever two states $u_1 \rhd a_1$ and $u_2 \rhd a_2$ are compared or joined, their partitionings $\Pi(u_1)$ and $\Pi(u_2)$ must agree. To this end, the coarsest partitioning $P := \{p_1 \cap p_2 \mid p_1 \in \Pi(u_1), p_2 \in \Pi(u_2)\}$ whose partitions can be merged to give either $\Pi(u_1)$ or $\Pi(u_2)$ is calculated. We then associate each partition $p \in P$ with a fresh flag f_p, thereby obtaining a new state $u_{12} = \bigcup_{p \in P}[f_p \mapsto p]$. Let $u_{12} = \mathit{common}(u_1, u_2)$ abbreviate this operation. Since u_{12} associates different (and possibly more) flags with its partitions than u_1 and u_2, the flags stored in a_1 and a_2 have to be adjusted. Thus, let $\mathit{trans}_{u_i}^{u_{12}}(f) := \{u_{12}(x) \mid x \in u_i^{-1}(f)\}$ denote the flags of those partitions in u_{12} whose union is associated with f in u_i. We transfer the value of f to the flags $\{f_1, \ldots, f_n\} \in \mathit{trans}_{u_i}^{u_{12}}(f)$ using the assignment $\mathit{adjOne}_{u_i}^{u_{12}}(f) := [\![f_1 := f]\!] \cdots [\![f_n := f]\!]$. The assignment for all partitions is then given by the composition $\mathit{adjust}_{u_i}^{u_{12}} := \bigcirc_{f \in \mathrm{img}(u_i)} \mathit{adjOne}_{u_i}^{u_{12}}(f)$.

$x \in \mathcal{X}$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
$u_1(x)$	f_1	f_1	f_1	f_1	f_2	f_2	f_2	f_2	f_3	f_3
$u_2(x)$	f_4	f_4	f_5	f_5	f_5	f_5	f_6	f_6	f_7	f_7
$u_{12}(x)$	f_8	f_8	f_9	f_9	f_{10}	f_{10}	f_{11}	f_{11}	f_{12}	f_{12}

Fig. 5. Partition $u_{12} = \mathit{common}(u_1, u_2)$ of Example 3.

Making two child states a_1 and a_2 compatible with u_{12} requires that the flags $\mathrm{img}(u_{12})$ are introduced, the renaming $\mathit{adjust}_{u_i}^{u_{12}}$ is applied, and that the now stale flags $\mathrm{img}(u_i)$ are removed. These operations are aggregated by the function $\mathit{compat}_{u_i}^{u_{12}} = \mathit{drop}_{\mathrm{img}(u_i)} \circ \mathit{adjust}_{u_i}^{u_{12}} \circ \mathit{add}_{\mathrm{img}(u_{12})}$.

Example 3. Consider the task of making two domains, $u_1 \rhd a_1$ and $u_2 \rhd a_2$ compatible where u_1 and u_2 are given by the first two rows in Fig. 5. First, the new partition $u_{12} = \mathit{common}(u_1, u_2)$ is calculated as shown in the last line of Fig. 5. In order to adjust, a_1 to be compatible with u_{12}, we compute $a_1' = \mathit{compat}_{u_1}^{u_{12}}(a_1) = \mathit{drop}_{\{f_1,f_2,f_3\}}(\mathit{adjust}_{u_1}^{u_{12}}(\mathit{add}_{\{f_8,\ldots f_{12}\}}(a_1)))$. The function $\mathit{adjust}_{u_1}^{u_{12}}$ expands to $\mathit{adjOne}_{u_1}^{u_{12}}(f_1) \cdots \mathit{adjOne}_{u_1}^{u_{12}}(f_3) = [\![f_8 = f_1, f_9 = f_1]\!] \cdot [\![f_{10} = f_2, f_{11} = f_2]\!] \cdot [\![f_{12} = f_3]\!]$. Computing $a_2' = \mathit{compat}_{u_2}^{u_{12}}(a_2)$ analogously suffices to perform any operation that requires $\chi(a_1') = \chi(a_2')$, such as $(u_1 \rhd a_1) \sqcup_{U(A)} (u_2 \rhd a_2) = u_{12} \rhd (a_1' \sqcup_A a_2')$.

This concludes the process of making domains compatible which allows us to associate a flag with a partition rather than a single variable. While tracking fewer flags improves performance, we now detail how precision can be improved.

3.3 Rescuing Relational Information

Tracking whether a set of variables is undefined is only useful if the content of undefined variables is replaced by other values that lead to less precision loss. In order to distinguish variables that are always undefined, we use a special flag f_{undef} whose value is always zero in the child domain. The variables $u^{-1}(f_{\mathit{undef}})$ associated with f_{undef} are omitted from the child domain. Due to this, computing the join of two states $(u_1 \rhd a_1) \sqcup_{U(A)} (u_2 \rhd a_2)$ requires that the variables $X_{12} = u_1^{-1}(f_{\mathit{undef}}) \setminus u_2^{-1}(f_{\mathit{undef}})$ that are undefined in a_1 but not in a_2 are added to a_1 before the child states a_i can be joined (and vice-versa). To this end, define a function $\mathit{copyAndPaste}_{D,X} : D \times D \to D$ with $r = \mathit{copyAndPaste}_{D,X}(s, a)$ such that variables X are copied from s into a, yielding r where $X \subseteq \chi(s)$, $\chi(a) \cap X = \emptyset$ and $\chi(r) = \chi(a) \cup X$. We illustrate *copyAndPaste* with an example:

Example 4. Suppose the following modified version of the introductory example is given where `rnd(0,10)` returns a number between 0 and 10:

```
1   int x,y;
2   if (rnd()) {
3     x = rnd(0,10);
4     y = x;
5   }
```

$$(u_1 \rhd a_1) \boxdot_{U(A)} (u_2 \rhd a_2) = \text{let for } i = 1,2 \quad (1)$$
$$X_i = u_i^{-1}(f_{undef})$$
$$u_i' = u_i[x \mapsto f_i]_{x \in X_i \setminus X_{3-i}} \text{ where } f_i \text{ fresh}$$
$$u_{12} = \mathit{common}(u_1', u_2')$$
$$a_i' = \mathit{copyAndPaste}_{A, X_{3-i} \setminus X_i}(a_{3-i}, a_i)$$
$$\text{in } u_{12} \rhd (\mathit{compat}_{u_1}^{u_{12}}(a_1') \boxdot_A \mathit{compat}_{u_2}^{u_{12}}(a_2'))$$

$$\mathit{add}_x(u \rhd a) = u[x \mapsto f_{undef}] \rhd a \quad (2)$$

$$\mathit{drop}_x(u \rhd a) = \text{if } u(x) = f_{undef} \text{ then } (u \setminus x) \rhd a \text{ else} \quad (3)$$
$$\text{if } |\{y \in \mathrm{dom}(u) \mid u(x) = u(y)\}| > 1$$
$$\text{then } (u \setminus x) \rhd \mathit{drop}_{A,x}(a)$$
$$\text{else } (u \setminus x) \rhd \mathit{drop}_{A,\{x,u(x)\}}(a) \quad (4)$$

$$[\![y := f(x_1, \ldots, x_n)]\!]_{U(A)} u \rhd a = \text{let } \Phi := \{u(x_1), \ldots u(x_n)\} \text{ in} \quad (5)$$
$$\text{if } f_{undef} \in \Phi \text{ then } \mathit{add}_{U(A),y}(\mathit{drop}_{U(A),y}(u \rhd a)) \text{ else}$$
$$\text{if } \Phi = \{f\} \text{ then } u[y \mapsto f] \rhd [\![y := f(x_1, \ldots, x_n);]\!]_A a$$
$$\text{else let } f_y \text{ fresh and } u' = u[y \mapsto f_y] \text{ in} \quad (6)$$
$$u' \rhd [\![y := f(x_1, \ldots, x_n);\ f_y := \sum_{f \in \Phi} f = |\Phi|]\!]_A a \quad (7)$$

$$[\![f(x_1, \ldots, x_n) \leq 0]\!]_{U(A)} u \rhd a = \text{let } \Phi := \{u(x_1), \ldots u(x_n)\} \text{ and } \psi = \sum_{f \in \Phi} f \text{ in} \quad (8)$$
$$u \rhd [\![f(x_1, \ldots, x_n) \leq 0; \psi = |\Phi|]\!]_A a \sqcup_A [\![\psi < |\Phi|]\!]_A a$$

Fig. 6. Transfer functions for binary operations $\boxdot = \sqcup, \sqsubseteq, \nabla$, and unary operations.

Consider analyzing this program with a state $u_1 \rhd a_1$ where $u_1 = [f_{undef} \mapsto \{x, y\}]$ and $a_1 = \{f_{undef} = 0\}$ is a convex polyhedron. Note that $\chi(a_1) = \{f_{undef}\}$ since the variables $x, y \in u_1^{-1}(f_{undef})$ are not stored in a_1 as explained above. The state at line 5 becomes $u_2 \rhd a_2$ where $u_2 = [f_{def} \mapsto \{x, y\}]$ and $a_2 = \{x = y, x \in [0, 10], f_{def} = 1\}$. The benefit of not storing x, y in a_1 is that they can be introduced using $a_1' = \mathit{copyAndPaste}_{A,\{x,y\}}(a_2, a_1) = \{x = y, x \in [0, 10], f_{undef} = 0\}$ that extracts all information over x, y in a_2 and adds it to a_1. In order to state that these variables are now explicitly stored in a_1', we rename f_{undef} to a new flag f_{xy}, yielding $u_2' \rhd a_2'$ with $u_2' = [f_{xy} \mapsto \{x, y\}]$ and $a_2' = \{x = y, x \in [0, 10], f_{xy} = 0\}$. Now the state after line 5 can be computed as $(u_1 \rhd a_1) \sqcup_{U(A)} (u_2' \rhd a_2') = u_{12} \rhd a_1' \sqcup_A a_2'$ where $u_{12} = u_2'$ and a_2' is a_2 in which f_{def} is renamed to f_{xy}. The result $a_1' \sqcup_A a_2' = \{x = y, x \in [0, 10], 0 \leq f_{xy} \leq 1\}$ retains the equality $x = y$, thereby improving the precision over copying intervals.

Figure 6 illustrates the implementation of the $\boxdot = \sqcup, \sqsubseteq, \nabla$ functions that use *copyAndPaste* on the child domain of type A. Here, we assume that $r = \mathit{copyAndPaste}_{D,X}(s, a)$ is defined as $r = s \sqcap_D \mathit{drop}_{D,\chi(a) \setminus X}(a)$ where $\sqcap_D$ is a

greatest lower bound on two abstract states that adds missing dimensions as needed. The idea is to remove all dimensions from a that should not be copied before merging the remaining relations over X into s using the meet $\sqcap_D$. For each binary operation $\boxdot$, Eq. 1 shows how the states are made compatible as described above before applying $\boxdot$ on the child domains.

Figure 6 also defines other transfer functions of the Undefined domain. Adding an unrestricted dimension x using *add* merely adds a mapping $f_{undef} \mapsto x$ to the undefined mapping (Eq. 2). Removing a variable x using *drop* needs to check if x is not stored in a (Eq. 3), or if it was the last variable in its partition (Eq. 4). Assigning to a variable y computes the set of flags Ψ that must be one to make the result defined (Eq. 5). If $f_{undef} \in \Psi$ then y is always undefined and executing the assignment on the child is not necessary. If a single flag f suffices to make y defined, y is added to the partition of f. In the general case, a new flag f_y is created (Eq. 6) that represents the validity of the new partition $\{y\}$ (Eq. 7).

Applying a test (Eq. 8) partitions the child state a into one where all variables occurring in the test are defined ($\psi = |\Phi|$) and one where they are possibly undefined ($\psi < |\Phi|$). Only in the former case, the test is applied.

Analogously to Lemma 2, the following lemma states that with the concrete domain C as its child domain, the Undefined domain is an abstraction of C.

Lemma 3. *With the proposed implementation, domain $U(C)$ is an abstraction of the concrete domain C.*

Proof. Beneath the choice of function *copyAndPaste* for adding variables, the implementation only differs from the semantics given in the last section by the more efficient, but otherwise equivalent handling of flags. It remains to show that adding dimensions via function *copyAndPaste* is valid, which holds because for each domain D, state a and set of variables X, function $\lambda s.\mathit{copyAndPaste}_{D,X}(s, a)$ is an X-Adder since $\mathit{drop}_X(\mathit{copyAndPaste}_X(s, a)) = a$.

The following theorem states that the given implementation of the Undefined domain is indeed a sound approximation of the concrete domain.

Theorem 2. *Let A be an abstraction of the concrete domain C. Then, with the proposed implementation, domain $U(A)$ is an abstraction of C.*

Proof. Analogously to Theorem 1, the claim follows from Lemma 3.

4 Applications of the Undefined Domain

We now illustrate the utility of the Undefined domain by using examples from the analysis of function calls and of heap-allocated memory.

4.1 Merging Calls to Functions

For the sake of limiting the memory consumption of an analyzer, it is desirable to merge the states of certain call sites of a function `f` into one. To this end,

```
main() {          a(int x) {        b(int y) {        f(int z) {
    a(0);             f(x);             f(y);             ...
    b(1);         }                 }                 }
}
```

Fig. 7. Function calls example.

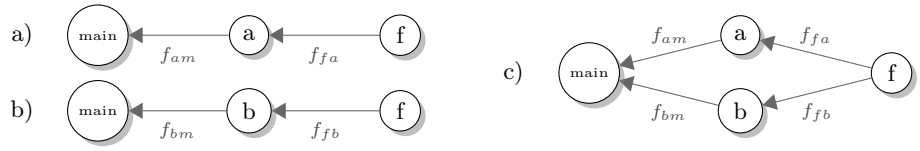

Fig. 8. Combining several call sites into one state.

we use a stack functor domain $G(S)$ (with child state S) that manages a set of stack frames. Here, G tracks one dedicated *active* stack frame that represents the currently executed function `f`. In order to track to which stack frame the analysis has to return to when leaving the current function, the state $g \in G$ is a directed graph with stack frames as nodes, where the more recently called function points to its caller. Consider for example the program in Fig. 7. Here, function `f` is called twice. First, it is called by function `a`, which in turn is called from `main`. Figure 8a) shows how the first call path via `a` forms a linked list of stack frames, say g_a. Figure 8b) shows the graph of stack frames for the second call to `f` via `b`, say g_b.

In order to combine two graphs g_a and g_b, we follow [10] in qualifying the graph edges by numeric flags, that is, numeric variables that can take on the values 0 or 1. Let $g_a \rhd s_a$ with $s_a = \{x = z = 0, f_{fa} = 1, f_{am} = 1\} \in S$ denote the abstract state (here $s_a \in S$ are convex polyhedra [4]) on entry to `f` for the path in Fig. 8a). In s_a, the flag f_{fa} has value one, indicating that the node (stack frame) of `a` is the predecessor of the node (stack frame) of `f`. Analogous for f_{am} that qualifies the edge between the stack frame of `f` and of `main`. Symmetrically, for the path shown in Fig. 8b) the state is $s_b = \{y = z = 1, f_{fb} = 1, f_{bm} = 1\}$.

The two graphs g_a and g_b are merged into the combined graph of stack frames g in Fig. 8c). In order to capture that the `b` node is not a predecessor of `f` in g_a, we add the flag $f_{fb} = 0$ to s_a and analogously we add $f_{bm} = 0$, yielding $s'_a = \{x = z = 0, f_{fa} = 1, f_{fb} = 0, f_{am} = 1, f_{bm} = 0\}$. Symmetrically, we enrich s_b to $s'_b = \{y = z = 1, f_{fa} = 0, f_{fb} = 1, f_{am} = 0, f_{bm} = 1\}$. Overall, we obtain the state $g \rhd s'_a \sqcup_S s'_b = g \rhd \{x = \top, y = \top, 0 \le z \le 1, z = f_{bm} = f_{fb} = 1 - f_{am} = 1 - f_{fa}\}$.

Note that all information within the stack frames, namely x and y is lost. The Undefined domain can improve this situation: we re-analyze the example using the domain $G(U(S))$. The net effect is that in the last step, instead of $g \rhd s'_a \sqcup_S s'_b$ we compute $g \rhd (u \rhd s'_a) \sqcup_{U(S)} (u \rhd s'_b)$ where $u \in U$ is the empty mapping (all variables are defined). By the definition of $\sqcup_{U(S)}$ the missing variable x is added to $u \rhd s'_b$ giving $u_b \rhd s'_b$ with $u_b = [x \mapsto f_{undef}]$ and, analogously, the left argument becomes $u_a \rhd s'_a$ with $u_a = [y \mapsto f_{undef}]$. Computing the join $u_a \rhd s'_a \sqcup_{U(S)} u_b \rhd s'_b$ makes the two undefined states u_a and u_b compatible to $u = [x \mapsto f_x, y \mapsto f_y]$. The numeric state s'_a is modified by adding $f_x = 1, f_y = 0$

a)

```
main() {
  int *p;
  if (rnd()) {
    p = malloc(4);
    *p = 3;
  } else
    p = NULL;
  if (p != NULL)
    assert *p == 3;
  return 0;
}
```

b)

```
struct point {int x; int y;} *p;
int n = rnd(0, 100);
for (int i=0; i<n; i++) {
  if (p == NULL)
    p = malloc(sizeof (*p));
  p->x = i;
  p->y = i;
}
if (p != NULL)
  assert p->x == p->y;
```

Fig. 9. Heap allocation examples

and copying $y = 1$ from s'_b whereas s'_b is modified by adding $f_x = 0, f_y = 1$ and copying $x = 0$ from s'_a. The state that `f` is analyzed with is thus $g \rhd u \rhd \{x = 0, y = 1, 0 \leq z \leq 1, z = f_x = f_{bm} = f_{fb} = 1 - f_y = 1 - f_{am} = 1 - f_{fa}\}$.

The benefit of the Undefined domain is thus that, upon returning from `f`, the content of the predecessor stack frames is still available since $x = 0, y = 1$ is retained in the join of the two call sites. Our analysis infers more intricate invariants if pointers are passed, since the flags of the Undefined domain form an equality relation with the points-to flags, as detailed in the next section.

4.2 Application to Heap Analysis

We now detail how the Undefined domain can improve precision in the analysis of programs that use dynamically allocated memory. To this end, consider the program in Fig. 9a) that contains the conditional statement of Fig. 1 of the introduction. After executing the then-branch of the first conditional, the state consists of a dynamically allocated memory region x that contains a single memory cell of value 3 and a variable p that holds a pointer to region x. We model pointer expressions as linear combinations of abstract addresses where the coefficients are numeric flags. For example, the expression $c \cdot \&x$ is a pointer to l-value x if $c = 1$ and a `NULL` pointer if $c = 0$ holds [10]. Thus, the state is $S_1 := \{p = c \cdot \&x, c = 1, x = 3\}$. Analogously, after executing the else-branch, the numeric state is just $S_2 := \{p = 0\}$.

In order to join the resulting states S_1 and S_2 of both branches, they are made compatible by extending state S_2: the numeric variable x is introduced and the pointer expression stored for p is extended with $c \cdot \&x$ with flag $c = 0$, resulting in state $S'_2 := \{p = c \cdot \&x, c = 0, x \in \mathbb{Z}\}$. Approximating the join $S_1 \sqcup S'_2$ in the Polyhedra domain yields a state $\{p = c \cdot \&x, c \in [0, 1], x \in \mathbb{Z}\}$, thus losing the information that $c = 1$ implies $x = 3$.

Using the Undefined domain, the state after the then-branch is $\emptyset \rhd S_1$ with child state S_1 and an empty mapping from program variables to flags. Similarly, the state after the else-branch is $\emptyset \rhd S_2$. When both states are made compatible, a new flag f is introduced in both states that indicates whether variable x is

	U	instructions	time	memory	variables	undef. flags	warnings
heap 1	✓	12	19	19.0	14	1	0
	✗	12	18	17.8	13	–	1
heap 2	✓	24	35	23.2	23	2	0
	✗	24	32	21.7	21	–	1
call stack 1	✓	114	450	42.0	50	2	0
	✗	76	377	41.9	48	–	7
call stack 2	✓	254	641	42.0	74	2	0
	✗	178	416	42.6	72	–	7
call stack 3	✓	153	718	42.4	66	4	0
	✗	76	422	41.9	48	–	7
call stack 4	✓	128	702	42.2	54	2	0
	✗	88	920	42.5	52	–	8
call stack 5	✓	173	1455	47.3	75	4	0
	✗	90	709	42.0	54	–	8

Fig. 10. Evaluation of our implementation

defined. The value of x is copied from S_1 to S_2 by the *copyAndPaste* operation. The joined state is now $\{f \mapsto \{x\}\} \rhd \{p = c \cdot \&x,\ c = f,\ f \in [0,1],\ x = 3\}$. Since the child domain expresses the invariant $f = c$, the information that $c = 1$ implies $x = 3$ is maintained. Analogously, $c = 0$ implies that `*p` is unbounded, reflecting the fact that an uninitialized memory location can hold any value and thereby guaranteeing that the analysis is still sound.

Figure 9b) shows a similar scenario where the Undefined domain is able to preserve the relation between the contents of a possibly nonexisting memory region during a fixpoint computation. There, a struct is allocated on the heap inside a loop and the variables in the struct are assigned the same value. After the loop, if the pointer to the struct exists, the program tests that the equality between the struct members still holds.

5 Implementation and Experimental Results

We evaluated the Undefined domain in our analyzer for machine code [9], using a domain stack $G(U(P(A(C(I)))))$ where G maintains stack frames and dynamically allocated memory, U is the Undefined domain as described in Sect. 3, and P tracks points-to sets of variables. The remaining domains are numeric; they track affine equalities A, congruences C and intervals I. In order to estimate the performance of the Undefined domain, we also evaluated the examples with domain stack $G(P(A(C(I))))$, that is, without domain U.

Our tool analyzes Intel assembler programs and translates each x86 instruction into a sequence of instructions over an intermediate representation (IR) [9]. The stack domain G recognizes function boundaries by observing the stack pointer whenever the control flow changes through a jump, call or return instruction. An x86 return instruction is translated into a read access to the previous stack frame in order to retrieve the return address and a jump to this address.

The Undefined domain is thereby key to infer a precise address since, for Fig. 8, stack frames a and b are both read and joined before the jump is executed.

Figure 10 shows the experimental results. Column U indicates whether the Undefined domain is used, followed by the number of machine instructions in the program that were analyzed; columns *time* and *memory* show the runtime in milliseconds and the memory consumption in megabytes, averaged over several runs on a 3.2 GHz Core i7 Linux machine. The next column shows the total number of variables tracked, followed by the number of flag variables used by the Undefined domain and the number of warnings emitted by the analyzer.

The first two lines show the heap example from Fig. 9a) that has been discussed in Sect. 4.2. Our implementation using the Undefined domain is able to verify the assertion in the program. Without the Undefined domain it raises a warning as the value of the heap allocated variable is lost. The next two lines show the heap example from Fig. 9b) where our analysis is able to verify an assertion in the program only when using the Undefined domain as the relational information between the struct members is lost otherwise.

Next follows the call stack example of Sect. 4.1, followed by variations with more functions and call paths. Call stack examples 4 and 5 differ in that they use pointers to stack variables to pass parameter values. Note that the call stack examples exhibited shorter runtimes without the Undefined domain, because precision loss made it impossible to resolve the return addresses, so that the examples could only partially be analyzed. This is reflected in the number of analyzed instructions. For the same reason the number of total variables in the call stack of example 3 and 5 without the domain are much lower than with the Undefined domain. The examples show that the additional variables that are necessary as flags for the Undefined domain are only few compared to the total number of variables in the program.

6 Related Work

We addressed the challenge of tracking the content of memory that does not exist in all traces. Many existing analyses use some ad-hoc methods to approximate what we have put on a sound mathematical basis: the ability to store both, precise and undefined values for variables in a single state. For instance, recency abstraction [1] implicitly retains the defined value when the state is joined. When a purely logic description is used [7, 8], the distinction between defined and undefined content is simply expressed using disjunction. In Astrée [2], disjunction is expressed using the decision tree domain that tracks two separate child domains depending on the value of a flag. The effect is similar to standard path-sensitive analyses in that tracking two states duplicates analysis time. More sophisticated analyses merge states on different paths if a finite abstraction determines that they are similar [5]. Future work will determine whether this technique can be implemented as a combinator in our domain stack.

The Undefined domain partially allows the encoding of conditional invariants. While this problem has been studied for logical domains [6], we provide a solution

that enables existing numeric domains to infer certain conditional invariants, e.g. those guarded by the existence of objects. For overly complex invariants, our approach exploits the ability of numeric domains to gradually lose precision.

7 Conclusion

We addressed the task of storing a single state in cases where a piece of memory is non-existent in some of the traces. We introduced a generic functor domain that generalizes this approach to existent memory regions with undefined content. We illustrated the power of this domain by defining a specific instance, namely the Undefined domain, that improves precision in common program analysis tasks. Its novel copy-and-paste operation even retains relational information.

References

1. G. Balakrishnan and T. Reps. Recency-Abstraction for Heap-Allocated Storage. In K. Yi, editor, *Static Analysis Symposium*, volume 4134 of *LNCS*, pages 221–239, Seoul, Korea, 2006. Springer.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation*, San Diego, California, USA, June 2003. ACM.
3. P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In M. Okada and I. Satoh, editors, *Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 272–300, Tokyo, Japan, December 2006. Springer.
4. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Principles of Programming Languages*, pages 84–97, Tucson, Arizona, USA, January 1978. ACM.
5. M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. *ACM SIGPLAN Notices*, 37(5):57, May 2002.
6. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles of Progamming Languages*, pages 235–246, San Diego, California, USA, January 2008. ACM.
7. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, 2002. IEEE.
8. M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
9. A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M.Pinzger and D. Poshyvanyk, editors, *Working Conference on Reverse Engineering*, Limerick, Ireland, October 2011. IEEE.
10. H. Siegel and A. Simon. FESA: Fold- and Expand-based Shape Analysis. In *Compiler Construction*, volume 7791 of *LNCS*, pages 82–101, Rome, Italy, March 2013. Springer.
11. A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis Symposium*, LNCS, pages 366–382, London, UK, 1996. Springer.