Static Program Analysis via 3-Valued Logic*

Thomas Reps[1], Mooly Sagiv[2], and Reinhard Wilhelm[3]

[1] Comp. Sci. Dept., University of Wisconsin; reps@cs.wisc.edu
[2] School of Comp. Sci., Tel Aviv University; msagiv@post.tau.ac.il
[3] Informatik, Univ. des Saarlandes;wilhelm@cs.uni-sb.de

Abstract. This paper reviews the principles behind the paradigm of "abstract interpretation via 3-valued logic," discusses recent work to extend the approach, and summarizes ongoing research aimed at overcoming remaining limitations on the ability to create program-analysis algorithms fully automatically.

1 Introduction

Static analysis concerns techniques for obtaining information about the possible states that a program passes through during execution, without actually running the program on specific inputs. Instead, static-analysis techniques explore a program's behavior for *all* possible inputs and *all* possible states that the program can reach. To make this feasible, the program is "run in the aggregate"—i.e., on abstract descriptors that represent collections of many states. In the last few years, researchers have made important advances in applying static analysis in new kinds of program-analysis tools for identifying bugs and security vulnerabilities [1–7]. In these tools, static analysis provides a way in which properties of a program's behavior can be verified (or, alternatively, ways in which bugs and security vulnerabilities can be detected). Static analysis is used to provide a safe answer to the question "Can the program reach a bad state?"

Despite these successes, substantial challenges still remain. In particular, pointers and dynamically-allocated storage are features of all modern imperative programming languages, but their use is error-prone:

- Dereferencing NULL-valued pointers and accessing previously deallocated storage are two common programming mistakes.
- Pointers and dynamically-allocated storage allow a program to build up complex graph data structures. Even when some simple data structure is intended, one or more incorrect assignments to pointers, or indirect assignments through pointers, can cause bugs whose symptoms are hard to diagnose.

Because tools for finding bugs and detecting security vulnerabilities need answers to questions about pointer variables, their contents, and the structure of the heap,[4] the usage of pointers in programs is a major obstacle to the goal of addressing software reliability by means of static analysis. In particular, the effects of assignments through pointer variables and pointer-valued fields make it hard to determine the aliasing relationships among different pointer expressions in a program. When less precise pointer information is available, the effectiveness of static techniques decreases.

Although much work has been done on algorithms for flow-insensitive points-to analysis [8–10] (including algorithms that exhibit varying degrees of context-sensitivity [11–15]), all of this work uses a very simple abstraction of heap-allocated storage: *All nodes allocated at site s are folded together into a single summary node n_s.* Such an approach has rather severe consequences for precision. If allocation site s is in a

* Supported by ONR contract N00014-01-1-0796, the Israel Science Foundation, and the A. von Humboldt Foundation.

[4] The term "heap" refers to the collection of nodes in, and allocated from, the free-storage pool.

loop, or in a function that is called more than once, then s can allocate multiple nodes with different addresses. A points-to fact "`p` points to n_s" means that program variable `p` may point to *one* of the nodes that n_s represents. For an assignment of the form `p->selector1 = q`, points-to-analysis algorithms are ordinarily forced to perform a "weak update": that is, selector edges emanating from the nodes that `p` points to are *accumulated*; the abstract execution of an assignment to a field of a summary node cannot "kill" the effects of a previous assignment because, in general, only *one* of the nodes that n_s represents is updated on each concrete execution of the assignment statement.

Such imprecisions snowball as additional weak updates are performed (e.g., for assignment statements of the form `r->selector2 = p->selector1`), and the use of a flow-insensitive algorithm exacerbates the problem. Consequently, most of the literature on points-to analysis leads to almost no useful information about the structure of the heap. One study [16] of the characteristics of the results obtained using one of the flow-insensitive points-to-analysis algorithms reports that

> Our experiments show that in every points-to graph, there is a single node (the "blob") that has a large number of outgoing flow edges. In every graph, the blob has an order of magnitude more outgoing edges than any other node.

Such imprecision, in turn, leads to overly pessimistic assessments of the program's behavior. Moreover, most of the representations of pointer relationships that have been proposed in the literature on points-to analysis cannot express even as simple a fact as "`x` points to an acyclic list". Such representations are unable to confirm behavioral properties, such as (i) when the input to a list-insert program is an acyclic list, the output is an acyclic list, and (ii) when the input to a list-reversal program that uses destructive-update operations is an acyclic list, the output is an acyclic list. Instead, most points-to-analysis algorithms will report that a possibly cyclic structure can arise. For programs that use two lists, most points-to-analysis algorithms will report that at the end of the program the two lists might share list elements (even when, in fact, the two lists must always remain disjoint).

The failings of conventional pointer-analysis algorithms discussed above are just symptomatic of a more general problem: in general, tools for finding bugs and detecting security vulnerabilities need answers to questions about a wide variety of behavioral properties; these questions can only be answered by tracking relationships among a program's runtime entities, and in general the number of such entities has no fixed upper bound. Moreover, the nature of the relationships that need to be tracked depends on both the program being analyzed and the queries to be answered.

The aim of our work [17] has been to create a *parametric framework for program analysis* that addresses these issues. A parametric framework is one that can be instantiated in different ways to create different program-analysis algorithms that provide answers to different questions, with varying degrees of efficiency and precision. The key aspect of our approach is the way in which it makes use of 2-valued and 3-valued logic: 2-valued and 3-valued *logical structures*—i.e., collections of predicates—are used to represent concrete and abstract stores, respectively; individuals represent entities such as memory cells, threads, locks, etc.; unary and binary predicates encode the contents of variables, pointer-valued structure fields, and other aspects of memory states; and first-order formulas with transitive closure are used to specify properties such as sharing, cyclicity, reachability, etc. Formulas are also used to specify how the store is affected by the execution of the different kinds of statements in the programming language.

The analysis framework can be instantiated in different ways by varying the predicate symbols of the logic, and, in particular, by varying which of the unary predicates control how nodes are folded together (this is explained in more detail in Sect. 2). The specified set of predicates determines the set of properties that will be tracked, and consequently what properties of stores can be discovered to hold at different points in the program by the corresponding instance of the analysis.

As a methodology for verifying properties of programs, the advantages of the 3-valued-logic approach are:

1. No loop invariants are required.
2. No theorem provers are involved, and thus every abstract execution step must terminate.
3. The method is based on abstract interpretation [18], and satisfies conditions that guarantee that the entire process always terminates.
4. The method applies to programs that manipulate pointers and heap-allocated data structures. Moreover, analyses are capable of performing *strong updates* during the abstract execution of an assignment to a pointer-valued field.
5. The method eliminates the need for the user to write the usual proofs required with abstract interpretation—i.e., to demonstrate that the abstract descriptors that the analyzer manipulates correctly model the actual heap-allocated data structures that the program manipulates.

A prototype implementation that implements this approach has been created, called TVLA (**T**hree-**V**alued-**L**ogic **A**nalyzer) [19, 20].

Points (1) and (2) may seem counterintuitive, given that we work with an undecidable logic (first-order logic plus transitive closure—see footnote 7), but they are really properties shared by any verification method that is based on abstract interpretation, and hence are consequences of point (3). Points (4) and (5) may be equally surprising—even to many experts in the field of static analysis—but are key aspects of this approach:

- Point (4) has a *fundamental effect on precision*. In particular, our approach is capable of confirming the behavioral properties mentioned earlier, i.e., (i) when the input to a list-insert program is an acyclic list, the output is an acyclic list, and (ii) when the input to a list-reversal program that uses destructive-update operations is an acyclic list, the output is an acyclic list. In addition, when a program uses multiple lists that always remain disjoint, our approach can often confirm that fact.
- Point (5) is one of the keys for *making the approach accessible for users*. With the methodology of abstract interpretation, it is often a difficult task to obtain an appropriate abstract semantics; abstract-interpretation papers often contain complicated proofs to show that a given abstract semantics is sound with respect to a given concrete semantics. With our approach, this is not the case: the abstract semantics falls out automatically from a specification of the concrete semantics (which has to be provided in any case whenever abstract interpretation is employed); the soundness of *all* instantiations of the framework follows from a single meta-theorem ([17, Theorem 4.9]).

The remainder of the paper is organized as follows: Sect. 2 summarizes the framework for static analysis from [17]. Sect. 3 describes several applications and extensions. Sect. 4 discusses related work.

2 The Use of Logic for Program Analysis

Modeling and abstracting the heap with logical structures. In the static-analysis framework defined in [17], concrete memory configurations—or *stores*—are modeled by logical structures. A logical structure is associated with a vocabulary of predicate symbols (with given arities): $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$ is a finite set of predicate symbols, where $\mathcal{P}_k$ denotes the set of predicate symbols of arity k (and $eq \in \mathcal{P}_2$). A logical structure supplies a predicate for each of the vocabulary's predicate symbols. A

```
typedef struct node {
  int data;
  struct node *n;
} *List;
```

(a)

Predicate	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same memory cell?
$q(v)$	Does pointer variable `q` point to memory cell v?
$n(v_1, v_2)$	Does the `n`-field of v_1 point to v_2?
$dle(v_1, v_2)$	Is the `data`-field of v_1 less than or equal to that of v_2?

(b)

Table 1. (a) Declaration of a linked-list datatype in C. (b) Core predicates used for representing the stores manipulated by programs that use type `List`. (We write predicate names in *italics* and code in `typewriter` font.)

concrete store is modeled by a 2-valued logical structure for a fixed vocabulary $\mathcal{C}$ of *core predicates*. Core predicates are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. Tab. 1 gives the definition of a C linked-list datatype, and lists the predicates that would be used to represent the stores manipulated by programs that use type `List`, such as the store shown in Fig. 1. 2-valued logical structures then represent memory configurations: the individuals are the set of memory cells; a nullary predicate represents a Boolean variable of the program; a unary predicate represents either a pointer variable or a Boolean-valued field of a record; and a binary predicate represents a pointer field of a record.[5]

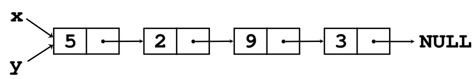


Fig. 1. A possible store, consisting of a four-node linked list pointed to by `x` and `y`.

The 2-valued structure S, shown in the upper-left-hand corner of Fig. 2, encodes the store of Fig. 1. S's four individuals, u_1, u_2, u_3, and u_4, represent the four list cells.

The following graphical notation is used for depicting 2-valued logical structures:

- An individual is represented by a circle with its name inside.
- A unary predicate p is represented by having a solid arrow from p to each individual u for which $p(u) = 1$, and by the absence of a p-arrow to each individual u' for which $p(u') = 0$. (If predicate p is 0 for all individuals, p is not shown.)
- A binary predicate q is represented by a solid arrow labeled q between each pair of individuals u_i and u_j for which $q(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u'_i and u'_j for which $q(u'_i, u'_j) = 0$.

Thus, in structure S, pointer variables `x` and `y` point to individual u_1, whose `n`-field points to individual u_2; pointer variables `t` and `e` do not point to any individual.

Often we only want to use a restricted class of logical structures to encode stores. To exclude structures that do not represent admissible stores, integrity constraints can be imposed. For instance, the predicate $x(v)$ of Fig. 2 captures whether pointer variable `x` points to memory cell v; x would be given the attribute "unique", which imposes the integrity constraint that $x(v)$ can hold for at most one individual in any structure.

The concrete operational semantics of a programming language is defined by specifying a structure transformer for each kind of edge that can appear in a control-flow graph. Formally, the structure transformer τ_e for edge e is defined using a collection of

[5] To simplify matters, our examples do not involve modeling numeric-valued variables and numeric-valued fields (such as `data`). It is possible to do this by introducing other predicates, such as the binary predicate dle (which stands for "`data` less-than-or-equal-to") listed in Tab. 1; dle captures the relative order of two nodes' `data` values. Alternatively, numeric-valued entities can be handled by combining abstractions of logical structures with previously known techniques for creating numeric abstractions [21].

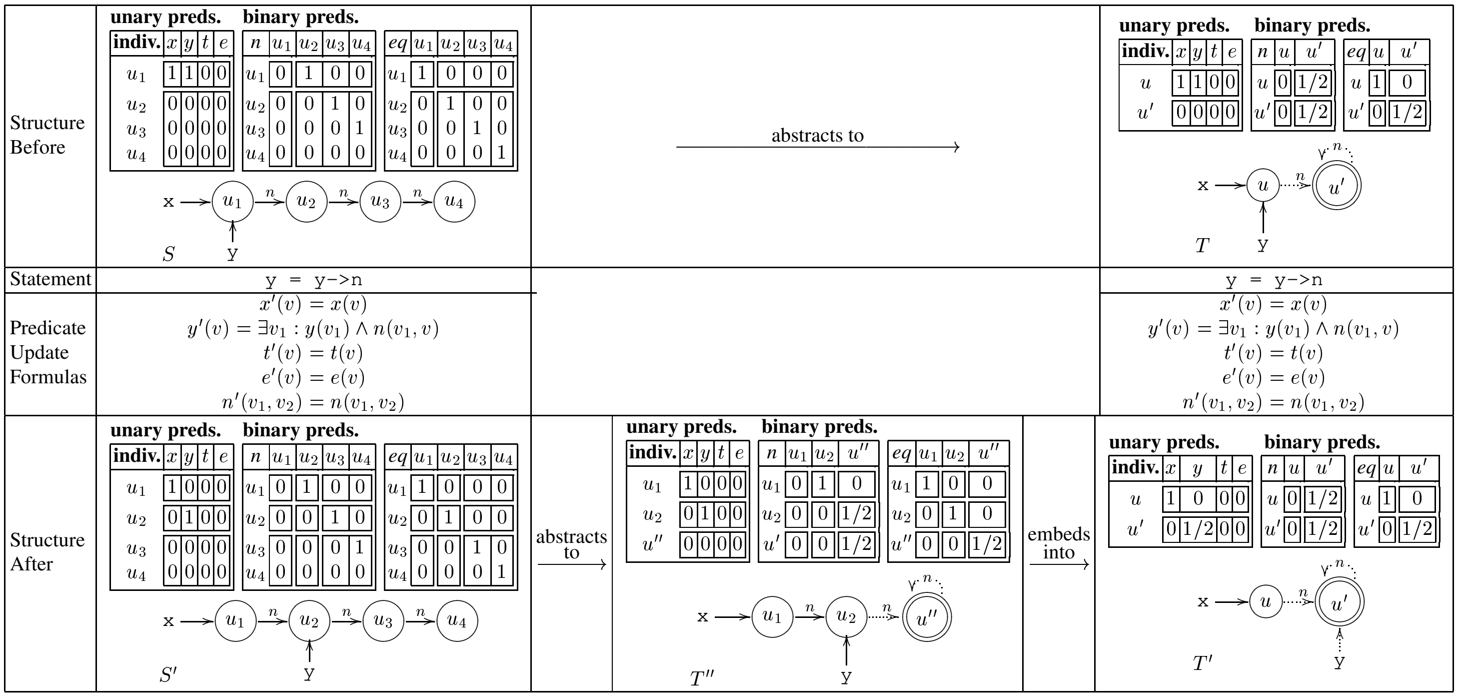


Fig. 2. The top row illustrates the abstraction of 2-valued structure S to 3-valued structure T with $\{x, y, t, e\}$-abstraction. The boxes in the tables of unary predicates indicate how individuals are grouped into equivalence classes; the boxes in the tables for n and eq indicate how the "truth-blurring quotients" are performed. The commutative diagram illustrates the relationship between (i) the transformation on 2-valued structures (defined by predicate-update formulas) for the concrete semantics of `y = y->n`, (ii) abstraction, and (iii) a sound abstract semantics for `y = y->n` that is obtained by using the same predicate-update formulas to transform 3-valued structures.

predicate-update formulas, $c(v_1, \ldots, v_k) = \tau_{c,e}(v_1, \ldots, v_k)$, one for each core predicate $c \in \mathcal{P}_k$ (e.g., see [17]). These define how the core predicates of a logical structure S that arises at the source of e are transformed to create structure S' at the target of e; they define the value of predicate c in S'—denoted by c' in the update formulas of Fig. 2—as a function of predicates in S. Edge e may optionally have a *precondition formula*, which filters out structures that should not follow the transition along e.

Canonical abstraction. To create abstractions of 2-valued logical structures (and hence of the stores that they encode), we use the related class of 3-valued logical structures over the same vocabulary. In 3-valued logical structures, a third truth value, denoted by $1/2$, is introduced to denote uncertainty: in a 3-valued logical structure, the value $p(\vec{u})$ of predicate p on a tuple of individuals $\vec{u}$ is allowed to be $1/2$.

Definition 1. The truth values 0 and 1 are *definite values*; $1/2$ is an *indefinite value*. For $l_1, l_2 \in \{0, 1/2, 1\}$, the *information order* is defined as follows: $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = 1/2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.

The abstract stores used for program analysis are 3-valued logical structures that, by the construction discussed below, are *a priori* of bounded size. In general, each 3-valued logical structure corresponds to a (possibly infinite) set of 2-valued logical structures. Members of these two families of structures are related by *canonical abstraction*.

The principle behind canonical abstraction is illustrated in the top and bottom rows of Fig. 2, which show how 2-valued structures S and S' are abstracted to 3-valued structures T and T'', respectively. The abstraction function is determined by a subset $\mathcal{A}$ of the unary predicates. The predicates in $\mathcal{A}$ are called the *abstraction predicates*. Given $\mathcal{A}$, the act of applying the corresponding abstraction function is called $\mathcal{A}$-*abstraction*. The canonical abstraction illustrated in Fig. 2 is $\{x, y, t, e\}$-abstraction.

Abstraction is driven by the values of the "vector" of abstraction predicates for each individual w—i.e., for S, by the values $x(w)$, $y(w)$, $t(w)$ and $e(w)$, for $w \in \{u_1, u_2, u_3, u_4\}$—and, in particular, by the equivalence classes formed from the individuals that have the same vector of values for their abstraction predicates. In S, there are two such equivalence classes: (i) $\{u_1\}$, for which x, y, t, and e are 1, 1, 0, and 0, respectively, and (ii) $\{u_2, u_3, u_4\}$, for which x, y, t, and e are all 0. (The boxes in the table of unary predicates for S show how individuals of S are grouped into two equivalence classes.) All of the members of each equivalence class are mapped to the same individual of the 3-valued structure. Thus, all members of $\{u_2, u_3, u_4\}$ from S are mapped to the same individual in T, called u';[6] similarly, all members of $\{u_1\}$ from S are mapped to the same individual in T, called u.

For each non-abstraction predicate p^S of 2-valued structure S, the corresponding predicate p^T in 3-valued structure T is formed by a "truth-blurring quotient". The value for a tuple $\vec{u}_0$ in p^T is the join ($\sqcup$) of all p^S tuples that the equivalence relation on individuals maps to $\vec{u}_0$. For instance,

- In S, $n^S(u_1, u_1)$ equals 0; therefore, the value of $n^T(u, u)$ is 0.
- In S, $n^S(u_2, u_1)$, $n^S(u_3, u_1)$, and $n^S(u_4, u_1)$ all equal 0; therefore, the value of $n^T(u', u)$ is 0.
- In S, $n^S(u_1, u_3)$ and $n^S(u_1, u_4)$ both equal 0, whereas $n^S(u_1, u_2)$ equals 1; therefore, the value of $n^T(u, u')$ is $1/2$ ($= 0 \sqcup 1$).

[6] The names of individuals are completely arbitrary: what distinguishes u' is the value of its vector of abstraction predicates.

- In S, $n^S(u_2, u_3)$ and $n^S(u_3, u_4)$ both equal 1, whereas $n^S(u_2, u_2)$, $n^S(u_2, u_4)$, $n^S(u_3, u_2)$, $n^S(u_3, u_3)$, $n^S(u_4, u_2)$, $n^S(u_4, u_3)$, and $n^S(u_4, u_4)$ all equal 0; therefore, the value of $n^T(u', u')$ is $1/2$ ($= 0 \sqcup 1$).

In the upper-left-hand corner of Fig. 2, the boxes in the tables for predicates n and eq indicate these four groupings of values.

In a 2-valued structure, the eq predicate represents the equality relation on individuals. In general, under canonical abstraction some individuals "lose their identity" because of uncertainty that arises in the eq predicate. For instance, $eq^T(u, u) = 1$ because u in T represents a single individual of S. On the other hand, u' represents three individuals of S and the quotient operation causes $eq^T(u', u')$ to have the value $1/2$. An individual like u' is called a *summary individual*.

A 3-valued logical structure T is used as an abstract descriptor of a set of 2-valued logical structures. In general, a summary individual models a *set* of individuals in each of the 2-valued logical structures that T represents. The graphical notation for 3-valued logical structures (cf. structure T of Fig. 2) is derived from the one for 2-valued structures, with the following additions:

- Individuals are represented by circles containing their names. (In Figs. 3 and 4, we also place unary predicates that do not correspond to pointer-valued program variables inside the circles.)
- A summary individual is represented by a double circle.
- Unary and binary predicates with value $1/2$ are represented by dotted arrows.

Thus, in every concrete structure $\tilde{S}$ that is represented by abstract structure T of Fig. 2, pointer variables `x` and `y` definitely point to the concrete node of $\tilde{S}$ that u represents. The `n`-field of that node may point to one of the concrete nodes that u' represents; u' is a summary individual, i.e., it may represent more than one concrete node in $\tilde{S}$. Possibly there is an `n`-field in one or more of these concrete nodes that points to another of the concrete nodes that u' represents, but there cannot be an `n`-field in any of these concrete nodes that points to the concrete node that u represents.

Note that 3-valued structure T also represents

- the acyclic lists of length 3 or more that are pointed to by `x` and `y`.
- the cyclic lists of length 3 or more that are pointed to by `x` and `y`, such that the back-pointer is not to the head of the list, but to the second, third, or later element.
- some additional memory configurations with a cyclic or acyclic list pointed to by `x` and `y` that also contain some garbage cells that are not reachable from `x` and `y`.

That is, T is a finite representation of an infinite set of (possibly cyclic) concrete lists, each of which may also be accompanied by some unreachable cells. Later in this section, we discuss options for fine-tuning an abstraction. In particular, we will use canonical abstraction to define an abstraction in which the acyclic lists and the cyclic lists are mapped to different 3-valued structures (and in which the presence or absence of unreachable cells is readily apparent).

Canonical abstraction ensures that each 3-valued structure has an *a priori* bounded size, which guarantees that a fixed-point will always be reached by an iterative static-analysis algorithm. Another advantage of using 2- and 3-valued logic as the basis for static analysis is that the language used for extracting information from the concrete world and the abstract world is identical: *every* syntactic expression—i.e., every logical formula—can be interpreted either in the 2-valued world or the 3-valued world.[7]

---

[7] Formulas are first-order formulas with transitive closure: a *formula* over the vocabulary $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$ is defined as follows (where $p^*(v_1, v_2)$ stands for the reflexive transitive

The consistency of the 2-valued and 3-valued viewpoints is ensured by a basic theorem that relates the two logics. This explains Point (5) mentioned in Sect. 1: the method eliminates the need for the user to write the usual proofs required with abstract interpretation. Thanks to a single meta-theorem (the Embedding Theorem [17, Theorem 4.9]), which shows that information extracted from a 3-valued structure T by evaluating a formula φ is sound with respect to the value of φ in each of the 2-valued structures that T represents, an abstract semantics falls out automatically from the specification of the concrete semantics. In particular, the formulas that define the concrete semantics when interpreted in 2-valued logic define a sound abstract semantics when interpreted in 3-valued logic (see Fig. 2). Soundness of *all* instantiations of the analysis framework is ensured by the Embedding Theorem.

Program analysis via 3-valued logic. A run of the analyzer carries out an abstract interpretation to collect a set of 3-valued structures at each program point. This involves finding the least fixed-point of a certain set of equations. Because canonical abstraction ensures that each 3-valued structure has an *a priori* bounded size, there are only a finite number of sets of 3-valued structures. This guarantees that a fixed-point is always reached. The structures collected at program point P describe a superset of all the execution states that can occur at P. To determine whether a property always holds at P, one checks whether it holds in all of the structures that were collected there.

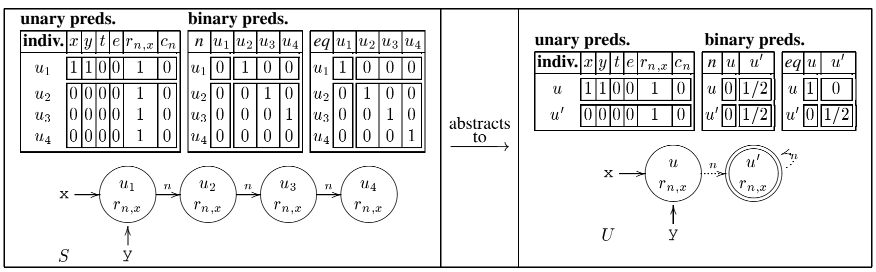


Fig. 3. The abstraction of 2-valued structure S to 3-valued structure U when we use $\{x, y, t, e, r_{n,x}, c_n\}$-abstraction. In contrast with T, T', and T'' from Fig. 2, U represents only acyclic lists of length 3 or more (with no garbage cells).

Instrumentation predicates. Unfortunately, unless some care is taken in the design of an analysis, there is a danger that as abstract interpretation proceeds, the indefinite value $1/2$ will become pervasive. This can destroy the ability to recover interesting information from the 3-valued structures collected (although soundness is maintained). A key role in combating indefiniteness is played by *instrumentation predicates*, which record auxiliary information in a logical structure. They provide a mechanism for the user to fine-tune an abstraction: an instrumentation predicate p of arity k, which is defined by a logical formula $\psi_p(v_1, \ldots, v_k)$ over the core predicate symbols, captures a property that each k-tuple of nodes may or may not possess. In general, adding additional instrumentation predicates refines the abstraction, defining a more precise analysis that is prepared to track finer distinctions among nodes. This allows more properties of the program's stores to be identified during analysis.

The introduction of unary instrumentation predicates that are then used as abstraction predicates provides a way to control which concrete individuals are merged together into summary nodes, and thereby to control the amount of information lost by

closure of $p(v_1, v_2)$):

$$\begin{array}{ll} p \in \mathcal{P}, \varphi \in \textit{Formulas}, & \varphi ::= \mathbf{0} \mid \mathbf{1} \mid p(v_1, \ldots, v_k) \mid (\neg\varphi_1) \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \\ v \in \textit{Variables} & \qquad\quad \mid (\exists v \colon \varphi_1) \mid (\forall v \colon \varphi_1) \mid p^*(v_1, v_2) \end{array}$$

abstraction. Instrumentation predicates that involve reachability properties, which can be defined using transitive closure, often play a crucial role in the definitions of abstractions (cf. Fig. 3). For instance, in program-analysis applications, reachability properties from specific pointer variables have the effect of keeping disjoint sublists or subtrees summarized separately. This is particularly important when analyzing a program in which two pointers are advanced along disjoint sublists. Tab. 2 lists some instrumentation predicates that are important for the analysis of programs that use type `List`.

p	**IntendedMeaning**	ψ_p
$t_n(v_1, v_2)$	Is v_2 reachable from v_1 along `n`-fields?	$n^*(v_1, v_2)$
$r_{n,x}(v)$	Is v reachable from pointer variable `x` along `n`-fields?	$\exists\, v_1 : x(v_1) \wedge t_n(v_1, v)$
$c_n(v)$	Is v on a directed cycle of `n`-fields?	$\exists\, v_1 : n(v, v_1) \wedge t_n(v_1, v)$

Table 2. Defining formulas of some commonly used instrumentation predicates. Typically, there is a separate predicate symbol $r_{n,x}$ for every pointer-valued variable `x`.

From the standpoint of the concrete semantics, instrumentation predicates represent cached information that could always be recomputed by reevaluating the instrumentation predicate's defining formula in the current store. From the standpoint of the abstract semantics, however, reevaluating a formula in the current (3-valued) store can lead to a drastic loss of precision. To gain maximum benefit from instrumentation predicates, an abstract-interpretation algorithm must obtain their values in some other way. This problem, the *instrumentation-predicate-maintenance problem*, is solved by incremental computation; the new value that instrumentation predicate p should have after a transition via abstract state transformer τ from state σ to σ' is computed incrementally from the known value of p in σ. An algorithm that uses τ and p's defining formula $\psi_p(v_1, \ldots, v_k)$ to generate an appropriate incremental predicate-maintenance formula for p is presented in [22].

The problem of automatically identifying appropriate instrumentation predicates, using a process of abstraction refinement, is addressed in [23]. In that paper, the input required to specify a program analysis consists of (i) a program, (ii) a characterization of the inputs, and (iii) a query (i.e., a formula that characterizes the intended output). This work, along with [22], provides a framework for eliminating previously required user inputs for which TVLA has been criticized in the past.

Other operations on logical structures. Thanks to the fact that the Embedding Theorem applies to any pair of structures for which one can be embedded into the other, most operations on 3-valued structures need not be constrained to manipulate 3-valued structures that are images of canonical abstraction. Thus, it is not necessary to perform canonical abstraction after the application of each abstract structure transformer. To ensure that abstract interpretation terminates, it is only necessary that canonical abstraction be applied as a widening operator somewhere in each loop, e.g., at the target of each back-edge in the control-flow graph.

Unfortunately, the simple abstract semantics obtained by applying predicate-update formulas to 3-valued structures can be very imprecise. For instance, in Fig. 2, `y = y->n` sets `y` to point to the next element in the list. In the abstract semantics, the evaluation in structure T of predicate-update formula $y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$ causes $y^{T'}(u')$ to be set to $1/2$. Consequently, all we can surmise in T' is that `y` may point to one of the cells that summary node u' represents. In contrast, the canonical abstraction of S' is T'', which demonstrates that the abstract domain is capable of representing a more precise abstract semantics than the T-to-T' transformation illustrated in Fig. 2.

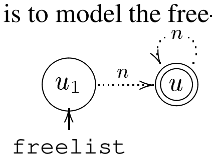
In [24], it is shown that for a Galois connection defined by abstraction function α and concretization function γ, the best abstract transformer for a concrete transformer τ, denoted by $\tau^\sharp$, can be expressed as: $\tau^\sharp = \alpha \circ \tau \circ \gamma$. This defines the limit of precision obtainable using a given abstract domain; however, it is a non-constructive definition: it does not provide an *algorithm* for finding or applying $\tau^\sharp$.

To help prevent an analysis from losing precision, several other operations on logical structures are used to implement a better approximation to the best transformer [17]:

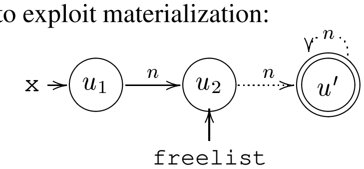
- *Focus* is an operation that can be invoked to elaborate a 3-valued structure—allowing it to be replaced by a set of more precise structures (not necessarily images of canonical abstraction) that represent the same set of concrete stores.
- *Coerce* is a clean-up operation that may "sharpen" a 3-valued structure by setting an indefinite value (1/2) to a definite value (0 or 1), or discard a structure entirely if the structure exhibits some fundamental inconsistency (e.g., it cannot represent any possible concrete store).

The transformers used in TVLA make use of *Focus*, *Coerce*, and incremental predicate-maintenance formulas to implement a better approximation to the best transformer than the T-to-T' transformation of Fig. 2. In particular, TVLA is capable of "materializing" non-summary nodes from summary nodes. For instance, given T, TVLA's transformer for `y = y->n` would create structure T'' (among others)—in essence, materializing u_2 out of u'. Materialization permits the abstract execution of an assignment to a pointer-valued field of a newly created non-summary node to perform a *strong update*.

Dynamically allocated storage. One way to model the semantics of `x = malloc()` is to model the free-storage list explicitly [22], so as to exploit materialization:




```
x = freelist;
freelist = freelist->n;
```



A `malloc` is modeled by advancing the pointer `freelist` into the list, and returning the memory cell that it formerly pointed to. A `free` is modeled by inserting, at the head of `freelist`'s list, the cell being deallocated.

3 Applications and Extensions

Interprocedural analysis. The application of canonical abstraction to interprocedural analysis of programs with recursion has been studied in both [25] and [26]. In [25], the main idea is to expose the runtime stack as an explicit "data structure" of the concrete semantics; that is, activation records are individuals, and suitable core predicates are introduced to capture how activation records are linked together to form a stack. Instrumentation predicates are used to record information about the calling context and the "invisible" copies of variables in pending activation records on the stack.

The analysis in [26] is based on logical structures over a doubled vocabulary $\mathcal{P} \uplus \mathcal{P}'$, where $\mathcal{P}' = \{p' \mid p \in \mathcal{P}\}$ and $\uplus$ denotes disjoint union. This approach creates a finite abstraction that relates the predicate values for an individual at the beginning of a transition to the predicate values for the individual at the end of the transition. Such two-vocabulary 3-valued structures are used to create a *summary transformer* for each procedure P, and the summary transformer is used at each call site at which P is called.

Checking multithreaded systems. In [27], it is shown how to apply 3-valued logic to the problem of checking properties of multithreaded systems. In particular, [27] addresses the problem of state-space exploration for languages, such as Java, that allow dynamic creation and destruction of an unbounded number of threads (as well as dynamic storage allocation and destructive updating of structure fields). Threads are modeled by individuals, which are abstracted using canonical abstraction—in this case, the collection of unary thread properties that hold for a given thread. The use of this naming scheme automatically discovers commonalities in the state space, but without relying on explicitly supplied symmetry properties, as in, for example, [28]. The analysis algorithm given in [27] builds and explores a 3-valued transition system on-the-fly. Unary core predicates are used to represent the program counter of each thread object; *Focus* is used to implement nondeterministic selection of a runable thread.

Numeric abstractions. The abstractions described in Sect. 2 are capable of representing pointer variables, their contents, and the structure of the heap, but have no direct way of representing the actual data items that are stored in the nodes of data structures. Recent work [21] has coupled canonical abstraction with a variety of previously known numeric abstractions: intervals, congruences, polyhedra [29], and various restrictions on polyhedral domains (such as difference constraints [30, 31] and 2-variable constraints [32]). These overapproximate the states that can arise in a program using sets of points in a k-dimensional space. However, when canonical abstraction is used to create bounded-size representations of memory configurations, the number of nodes in an abstract descriptor is different at different points in the program; for numeric abstractions, this means that the number of *axes* changes from program point to program point—i.e., there is not a fixed value of k. To capture numeric properties in such a *summarizing* framework, an analysis needs to be able to capture the relationships among values of *groups* of numeric objects, rather than relationships among values of *individual* numeric objects [21].

Best abstract transformers. As mentioned in Sect. 2, for a Galois connection (α, γ), a non-constructive definition of the best abstract transformer $\tau^\sharp$ for concrete transformer τ can be expressed as $\tau^\sharp = \alpha \circ \tau \circ \gamma$. This defines the limit of precision obtainable using a given abstract domain, but does not provide an *algorithm* for finding or applying $\tau^\sharp$. Graf and Saïdi [33] showed that decision procedures can be used to generate best abstract transformers for abstract domains that are finite Cartesian products of Boolean values. (The use of such domains is known as *predicate abstraction*.)

The ability to perform abstract interpretation using best abstract transformers could play a key role in combating indefiniteness in 3-valued structures. It would ensure that an abstract interpretation computes answers that are *precise up to the inherent limitations of the abstraction in use*. In recent work, we have made a start towards this goal. In particular, we have defined two approaches to computing best transformers for applications that use canonical abstraction [34, 35].

Applications. Some of the problems to which the 3-valued-logic approach has been applied include the following: In [36], TVLA was used to establish the partial correctness of bubble-sort and insert-sort routines for sorting linked lists. The abstraction-refinement method of [23] was used to extend this work to address stability and antistability properties of sorting routines. TVLA has also been used to demonstrate the total correctness of a mark-and-sweep garbage collector operating on an arbitrary heap. In Java, once an iterator object o_i is created for a collection o_c, o_i may be used only as long as o_c remains unmodified, not counting modifications made via o_i; otherwise a "concurrent modification exception" is thrown. In [37], TVLA was used to create a verification tool for establishing the absence of concurrent modification exceptions.

In the area of multithreaded systems, the 3-valued-logic approach has been used to establish the absence of deadlock for a dining-philosophers program that permits there to be an unbounded number of philosophers [27], as well as to establish the partial correctness of two concurrent queue algorithms; these results were obtained without imposing any *a priori* bound on the number of allocated objects and threads [38].

4 Related Work

Predicate abstraction. Canonical abstraction is sometimes confused with predicate abstraction, which has been used in a variety of systems [33, 39, 6, 40]. At one level, predicate abstraction and canonical abstraction use essentially the same mechanism:

- Predicate abstraction can be used to abstract a possibly-infinite transition system to a finite one: concrete states of the transition system are grouped into abstract states according to the values of a vector of properties. The transition relation is quotiented by the equivalence relation induced on concrete states.
- Canonical abstraction is used to abstract a possibly-infinite logical structure to a finite 3-valued one: concrete individuals are mapped to abstract individuals according to the values of a vector of unary abstraction predicates; all other predicates are quotiented by the equivalence relation induced on concrete individuals.

However, as used in [17], canonical abstraction is applied to encodings of stores as logical structures, and machinery is developed to use 3-valued structures to define a parametric abstract domain for abstract interpretation. Predicate abstraction has also been used to define a parametric abstract domain [41]. Thus, an alternative comparison criterion is to consider the relationship between the two parametric abstract domains:

- Predicate abstraction yields a parametric abstract domain based on finite Cartesian products of Booleans (i.e., nullary predicates). An abstract value consists of a finite set of finite-sized vectors of nullary predicates [41].
- Canonical abstraction yields a parametric abstract domain based on 3-valued logical structures. An abstract value consists of a finite set of finite-sized 3-valued structures [17].

A special case of canonical abstraction occurs when *no* abstraction predicates are used at all, in which case all individuals are collapsed to a single individual. When this is done, in almost all structures the only useful information remaining resides in the *nullary* core and instrumentation predicates. Predicate abstraction can be seen as going one step further, and retaining *only* the nullary predicates. From this point of view, canonical abstraction is *strictly more general* than predicate abstraction.

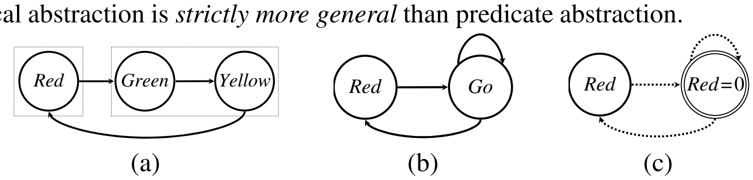


Fig. 4. (a) Transition diagram for a stoplight; (b) transition diagram abstracted via the method of [42] when *green* and *yellow* are mapped to *go*; (c) transition diagram abstracted via canonical abstraction, using *red*(v) as the only abstraction predicate.

Existential abstraction. Canonical abstraction is also related to the notion of *existential abstraction* used in [43, 42]. However, canonical abstraction yields 3-valued predicates and distinguishes summary nodes from non-summary nodes, whereas existential abstraction yields 2-valued predicates and does not distinguish summary nodes from non-summary nodes. Fig. 4 shows the transition diagram for a stoplight—an example used in [42]—abstracted via the method of [42] (Fig. 4(b)) and via canonical abstraction, using *red*(v) as the only abstraction predicate (Fig. 4(c)). With existential abstraction, soundness is preserved by restricting attention to universal formulas (formulas in ACTL*). With canonical abstraction, soundness is also preserved by switching logics, although in this case there is no syntactic restriction; we switch from 2-valued first-order logic to 3-valued first-order logic. An advantage of this approach is that if φ is any formula for a query about a concrete state, the same syntactic formula φ can be used to pose the same query about an abstract state.

One-sided versus two-sided answers. Most static-analysis algorithms provide 2-valued answers, but are *one-sided*: an answer is *definite* on one value and *conservative* on the other. That is, either 0 means 0, and 1 means "maybe"; or 1 means 1, and 0 means "maybe". In contrast, by basing the abstract semantics on 3-valued logic, definite truth and definite falseness can both be tracked, with $1/2$ capturing indefiniteness. (To determine whether a formula φ holds at P, it is evaluated in each of the structures that are collected at P. The answer is the *join* of these values.) This provides insight into the true nature of the one-sided approach. For instance, an analysis that is definite with respect to 1 is really a 3-valued analysis that conflates 0 and $1/2$ (and uses 0 in place of $1/2$). (It should be noted that with a two-sided analysis, the answers 0 and 1 are definite with respect to the concrete semantics *as specified*, which may itself overapproximate the behavior of the actual system being modeled.)

Acknowledgments. We thank our many students and collaborators.

References

1. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Softw. Tools for Tech. Transfer **2** (2000)
2. Wagner, D., Foster, J., Brewer, E., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Network and Dist. Syst. Security. (2000)
3. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Op. Syst. Design and Impl. (2000) 1–16
4. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Int. Conf. on Softw. Eng. (2000) 439–448
5. Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Software–Practice&Experience **30** (2000) 775–802
6. Ball, T., Rajamani, S.: The SLAM toolkit. In: Int. Conf. on Computer Aided Verif. Volume 2102 of Lec. Notes in Comp. Sci. (2001) 260–264
7. Chen, H., Wagner, D.: MOPS: An infrastructure for examining security properties of software. In: Conf. on Comp. and Commun. Sec. (2002) 235–244
8. Andersen, L.O.: Binding-time analysis and the taming of C pointers. In: Part. Eval. and Semantics-Based Prog. Manip. (1993) 47–58
9. Steensgaard, B.: Points-to analysis in almost-linear time. In: Princ. of Prog. Lang. (1996) 32–41
10. Das, M.: Unification-based pointer analysis with directional assignments. In: Prog. Lang. Design and Impl. (2000) 35–46
11. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: Prog. Lang. Design and Impl. (2000) 253–263
12. Cheng, B.C., Hwu, W.: Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In: Prog. Lang. Design and Impl. (2000) 57–69
13. Foster, J., Fähndrich, M., Aiken, A.: Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In: Static Analysis Symp. (2000) 175–198
14. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In: Prog. Lang. Design and Impl. (2004) To appear.
15. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: Prog. Lang. Design and Impl. (2004) To appear.
16. M.Das, Liblit, B., Fähndrich, M., Rehof, J.: Estimating the impact of scalable pointer analysis on optimization. In: Static Analysis Symp. (2001) 260–278

17. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. Trans. on Prog. Lang. and Syst. **24** (2002) 217–298
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: Princ. of Prog. Lang. (1977) 238–252
19. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Static Analysis Symp. (2000) 280–301
20. : (TVLA system) "http://www.math.tau.ac.il/∼rumster/TVLA/".
21. Gopan, D., DiMaio, F., N.Dor, Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Tools and Algs. for the Construct. and Anal. of Syst. (2004) 512–529
22. Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. In: European Symp. On Programming. (2003) 380–398
23. Loginov, A., Reps, T., Sagiv, M.: Abstraction refinement for 3-valued-logic analysis. Tech. Rep. 1504, Comp. Sci. Dept., Univ. of Wisconsin (2004)
24. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Princ. of Prog. Lang. (1979) 269–282
25. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Comp. Construct. Volume 2027 of Lec. Notes in Comp. Sci. (2001) 133–149
26. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. Tech. Rep. 1505, Comp. Sci. Dept., Univ. of Wisconsin (2004)
27. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: Princ. of Prog. Lang. (2001) 27–40
28. Emerson, E., Sistla, A.: Symmetry and model checking. In Courcoubetis, C., ed.: Int. Conf. on Computer Aided Verif. (1993) 463–478
29. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: Princ. of Prog. Lang. (1978)
30. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: Automatic Verification Methods for Finite State Systems. (1989) 197–212
31. Miné, A.: A few graph-based relational numerical abstract domains. In: Static Analysis Symp. (2002) 117–132
32. Simon, A., King, A., Howe, J.: Two variables per linear inequality as an abstract domain. In: Int. Workshop on Logic Based Prog. Dev. and Transformation. (2002) 71–89
33. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Int. Conf. on Computer Aided Verif. Volume 1254 of Lec. Notes in Comp. Sci. (1997) 72–83
34. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Verif., Model Checking, and Abs. Interp. (2004) 252–266
35. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Tools and Algs. for the Construct. and Anal. of Syst. (2004) 530–545
36. Lev-Ami, T., Reps, T., Sagiv, M., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: Int. Symp. on Softw. Testing and Analysis. (2000) 26–38
37. Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving specialized program analyses for certifying component-client conformance. In: Prog. Lang. Design and Impl. (2002) 83–94
38. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. In: Workshop on Software Model Checking. (2003)
39. Das, S., Dill, D., Park, S.: Experience with predicate abstraction. In: Int. Conf. on Computer Aided Verif., Springer-Verlag (1999) 160–171
40. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Princ. of Prog. Lang. (2002) 58–70
41. Ball, T., Podelski, A., Rajamani, S.: Boolean and Cartesian abstraction for model checking C programs. In: Tools and Algs. for the Construct. and Anal. of Syst. (2001) 268–283
42. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Int. Conf. on Computer Aided Verif. (2000) 154–169
43. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. Trans. on Prog. Lang. and Syst. **16** (1994) 1512–1542