Boolean Heaps

Andreas Podelski and Thomas Wies

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{podelski,wies}@mpi-inf.mpg.de

Abstract. We show that the idea of predicates on heap objects can be cast in the framework of predicate abstraction. This leads to an alternative view on the underlying concepts of three-valued shape analysis by Sagiv, Reps and Wilhelm. Our construction of the abstract post operator is analogous to the corresponding construction for classical predicate abstraction, except that predicates over objects on the heap take the place of state predicates, and boolean heaps (sets of bitvectors) take the place of boolean states (bitvectors). A program is abstracted to a program over boolean heaps. For each command of the program, the corresponding abstract command is effectively constructed by deductive reasoning, namely by the application of the *weakest precondition* operator and an entailment test. We thus obtain a symbolic framework for shape analysis.

1 Introduction

The transition graph of a program is formed by its states and the transitions between them. The idea of *predicate abstraction* [6] (used in a tool such as SLAM [2]) is to abstract a state by its evaluation under a number of given state predicates; each edge between two concrete states in the transition graph gives rise to an edge between the two corresponding abstract states. One thus abstracts the transition graph to a graph over abstract states.

For a program manipulating pointers, each state is represented by a *heap graph*. A heap graph is formed by the allocated objects in the heap and pointer links between them. The idea of *three-valued shape analysis* [13] is to apply to the heap graph the same abstraction that we have applied to the transition graph. One abstracts an object in the heap by its evaluation under a number of *heap predicates*; edges between concrete objects in the heap graph give rise to edges between the corresponding abstract objects. One thus abstracts a heap graph to a graph over abstract objects.

The analogy between predicate abstraction and the abstraction proposed in three-valued shape analysis is remarkable. It does not seem helpful, however, when it comes to the major challenge: how can one compute the abstraction of the transition graph when states are heap graphs and the abstraction is defined on objects of the heap graph? This paper answers a refinement of this question, namely whether the abstraction can be defined and computed in the formal setup and with the basic machinery of predicate abstraction.

Our technical contributions that are needed to accomplish this task are summarized as follows:

– We omit explicit edges between abstract objects in the abstract state, since we observe that one can also encode edge relations implicitly using appropriate heap predicates on objects. This makes it possible to define the abstract post operator by *local* updates of the values of heap predicates.
– We show that one can implement the abstraction by a simple source-to-source transformation of a pointer program to an abstract finite-state program which we call a *boolean heap program*. This transformation is analogous to the corresponding transformation in predicate abstraction, except that predicates over objects on the heap take the place of state predicates and boolean heaps (sets of bitvectors) take the place of boolean states (bitvectors).
– We formally identify the post operator of a boolean heap program as an abstraction of the best abstract post operator on an abstract domain of formulas. For each command of the program, the corresponding abstract command is constructed by the application of a *weakest precondition* operator on heap predicates and an entailment test (implemented by a syntactic substitution resp. by a call to a theorem prover).

Outline. In Section 2 we give related work; in particular, we summarize the key concepts of predicate abstraction. Section 3 gives the algorithmic description of our analysis. Section 4 defines the formal semantics of programs manipulating pointers. In Section 5 we give a theory of heap predicates that extends the notion of state predicates and state predicate transformers to predicates on heap objects and heap predicate transformers. Section 6 provides a formal definition of our analysis in the framework of abstract interpretation. In Section 7 we formally identify the abstract system described in Section 3 as a composition of additional abstraction functions with the best abstract post operator on our abstract domain. Section 8 concludes. Omitted proofs can be found in the extended version of this paper[1].

2 Related Work

In [13] Sagiv, Reps and Wilhelm describe a parametric framework to shape analysis based on three-valued logic. They abstract sets of states by three-valued logical structures. The abstraction is defined in terms of equivalence classes of objects in the heap that are induced by a finite set of predicates on heap objects. We use several ideas from this approach. In particular, there is a strong connection between their abstract domain and ours: a translation from three-valued logical structures, as they arise in [13], into formulas in first-order logic is given in [15]. Shape analysis constraints [10] extend this translation to a boolean algebra of state predicates that is isomorphic to the class of three-valued logical structures in [13]; our abstract domain is a fragment of shape analysis constraints.

[1] available on the web at `http://www.mpi-inf.mpg.de/~wies/papers/boolean-heaps-extended.pdf`

In [16] a symbolic algorithm is presented that can be used for shape analysis *à la* [13]. It is based on an *assume* operation that is implemented using a decision procedure. The *assume* operation allows inter-procedural shape analysis based on assume-guarantee reasoning. Moreover, *assume* can be instantiated to compute best abstraction functions, most-precise post operators, and the meet operation for abstract domains of three-valued logical structures. In our framework we do not depend on an intermediate representation of sets of states in terms of three-valued logical structures. We work exclusively on formulas.

PALE [12] is a Hoare-style system for the analysis of pointer programs that is based on weak monadic second order logic over trees. Its degree of automation is restricted, because loops in the program have to be manually annotated with loop invariants. Also the class of data structures that PALE is able to handle is restricted to graph types [9]. In our approach we synthesize loop invariants automatically. Furthermore, our analysis is not restricted *a priori* to a particular class of data structures; which data structures our analysis is able to treat only depends on the capabilities of the underlying theorem prover that is used to compute the abstraction.

Software model checkers such as SLAM [2] use predicate abstraction [6] to abstract the concrete transition system into a finite-state boolean program. A state of the resulting boolean program, i.e. an abstract state, is given by a bitvector over the abstraction predicates. Each transition of the concrete system gives rise to a corresponding simultaneous update of the predicate values in the boolean program.

General scheme

Concrete command:
c

State predicates:
$\mathcal{P} = \{p_1, \ldots, p_n\}$

Abstract boolean program:

var $p_1, \ldots, p_n$: boolean
for each $p_i \in \mathcal{P}$ **do**
  if $\mathsf{wp}^{\#}\, c\, p_i$ **then** $p_i := \mathsf{true}$
  else if $\mathsf{wp}^{\#}\, c\, (\neg p_i)$ **then** $p_i := \mathsf{false}$
  else $p_i := *$

Example

Concrete command:
var x : integer
$x := x + 1$

State predicates:
$p_1 \stackrel{def}{=} x = 0, \qquad p_2 \stackrel{def}{=} x > 0$

Abstract boolean program:

var p_1, p_2 : boolean
if false **then** $p_1 :=$ true
else if $p_1 \vee p_2$ **then** $p_1 :=$ false
else $p_1 := *$
if $p_1 \vee p_2$ **then** $p_2 :=$ true
else if $\neg p_1 \wedge \neg p_2$ **then** $p_2 :=$ false
else $p_2 := *$

Fig. 1. Construction of a boolean program from a concrete command via predicate abstraction. All predicates are updated simultaneously. The value '*' stands for nondeterministic choice.

Figure 1 shows the transformation of a concrete command to the corresponding predicate updates in the abstract boolean program. The actual abstraction step lies in the computation of $\mathsf{wp}^{\#}\ c\ p$ – the best boolean under-approximation (in terms of abstraction predicates) of the weakest precondition of predicate p and command c (for example false is the best under-approximation of $\mathsf{wp}^{\#}(x := x + 1)\ (x = 0)$ with respect to p_1 and p_2). One of the advantages of predicate abstraction is that the computation of this operator can be done *offline* in a pre-processing step (using a decision procedure or theorem prover). Therefore, one has a clear separation between the abstraction phase and the actual fixed point computation of the analysis.

There are several approaches that use classical predicate abstraction for shape analysis; see e.g. [5] and [1]. As discussed in [11], if one wants to gain the same precision with classical predicate abstraction as for the abstract domain proposed in [13] then in general one needs an exponential number of state predicates compared to the number of predicates on heap objects that are used in [13]. This seems to be the major drawback of using standard predicate abstraction for shape analysis. We solve this problem by combining the core ideas from both frameworks. In particular, we use Cartesian abstraction in a way that is reminiscent of the approach described in [3]. However, we restrict our attention to safety properties, whereas in [1] also liveness properties are considered.

3 Boolean Heap Programs

Our analysis proceeds as follows: (1) we choose a set of predicates over heap objects for the abstraction (defining the abstract domain); (2) we construct an abstract finite-state program in analogy to predicate abstraction (the abstract post operator); and (3) we apply finite-state model checking to the abstract program (the fixed point computation). In the following we explain in detail how the abstract domain and the construction of the abstract program look like.

For an abstract domain given by graphs over abstract objects it is difficult to compute the abstract post operator as an operation on the whole abstract state. Instead one would like to represent the abstract post operator corresponding to a pointer command by *local* updates. Local means that one updates each abstract object in isolation. However, pointer commands update pointer fields. The problem is: how can one account for the update of pointer fields by local updates on abstract objects?

The key idea is that we use a set of abstract objects to represent an abstract state, i.e. we omit edges between abstract objects. A state s is represented by a set of abstract objects, if every concrete object in s is represented by one abstract object in the set. Instead of having explicitly-encoded pointer relations in the abstract state, pointer information is implicitly encoded using appropriate predicates on heap objects for the abstraction. In particular, the presence or absence of an edge between two abstract objects can be encoded into heap predicates on objects. Adding these predicates to the set of abstraction predicates will preserve this information in the abstraction; see [14].

General scheme	**Example**
Concrete command:	Concrete command:
c	**var** x, y, z : list $x.next := y$
Unary heap predicates: $\mathcal{P} = \{p_1(v), \ldots, p_n(v)\}$	Unary heap predicates: $p_1(v) \stackrel{def}{=} x = v, \quad p_2(v) \stackrel{def}{=} y = z$ $p_3(v) \stackrel{def}{=} next(v) = z$

Boolean heap program:
var V : set of bitvectors over $\mathcal{P}$
for each $\overline{p} \in V$ **do**
  for each $p_i \in \mathcal{P}$ **do**
    if $\overline{p} \models \mathsf{hwp}^{\#}\ c\ p_i$
      then $\overline{p}.p_i := \mathsf{true}$
    else if $\overline{p} \models \mathsf{hwp}^{\#}\ c\ (\neg p_i)$
      then $\overline{p}.p_i := \mathsf{false}$
    else $\overline{p}.p_i := *$
	Boolean heap program:
	var V : set of bitvectors over $\{p_1, p_2, p_3\}$
	for each $\overline{p} \in V$ **do**
	  if $\overline{p} \models p_1$ **then** $\overline{p}.p_1 := \mathsf{true}$
	  else if $\overline{p} \models \neg p_1$ **then** $\overline{p}.p_1 := \mathsf{false}$
	  if $\overline{p} \models p_2$ **then** $\overline{p}.p_2 := \mathsf{true}$
	  else if $\overline{p} \models \neg p_2$ **then** $\overline{p}.p_2 := \mathsf{false}$
	  if $\overline{p} \models \neg p_1 \wedge p_3 \vee p_1 \wedge p_2$ **then**
	    $\overline{p}.p_3 := \mathsf{true}$
	  else if $\overline{p} \models \neg(\neg p_1 \wedge p_3 \vee p_1 \wedge p_2)$
	    then $\overline{p}.p_3 := \mathsf{false}$

Fig. 2. Transformation of a concrete command into a boolean heap program.

The set of abstract objects defining the abstract state is represented by a set of bitvectors over abstraction predicates; we call such a set of bitvectors a boolean heap. We abstract a pointer program by a *boolean heap program* as defined in Fig. 2. The construction naturally extends the one used in predicate abstraction which is given in Fig. 1. The difference is that a state of the abstract program is not given by a single bitvector, but by a set of bitvectors, i.e a boolean heap. Transitions in boolean heap programs change the abstract state via local updates on abstract objects ($\overline{p}.p_i := \mathsf{true}$) rather than global updates on the whole abstract state ($p_i := \mathsf{true}$). Consequently, we replace the abstraction of the weakest precondition operator on state predicates $\mathsf{wp}^{\#}$ by the abstraction of a weakest precondition operator on heap predicates $\mathsf{hwp}^{\#}$. While causing only a moderate loss of precision, this construction avoids the exponential blowup in the construction of the abstract program that occurs when standard predicate abstraction is used to simulate a graph based abstract domain with an appropriate set of state predicates.

In the rest of the paper we give a formal account of boolean heap programs. In particular, we make precise what it means to compute the operator $\mathsf{hwp}^{\#}$. Furthermore, we identify the post operator that corresponds to a boolean heap program as an abstraction of the best abstract post operator on boolean heaps. This way we identify the points in the analysis where we can lose precision.

4 Pointer Programs

We consider the language of pointer programs defined in Fig. 3. In order to highlight our main observations, we make several simplifications: (1) we do not model the program counter; (2) we do not consider allocation or deallocation of objects; and (3) we do not treat null pointers explicitly; in particular, we do not treat dereferences of null pointers. However, none of these simplifications imposes inherent restrictions of our results.

A state of the program is represented as a logical structure over the vocabulary of program variables *Var* and pointer fields *Field*. Since we do not treat allocation and deallocation of objects, we fix a set of objects *Heap* that is not changed during program execution and serves as the common universe of all program states. Therefore, a state degenerates to an interpretation function, i.e. a valuation of program variables to elements of *Heap* and pointer fields to total functions over *Heap*. Note that we define states as a Cartesian product of interpretation functions, but for notational convenience we implicitly project to the appropriate component function when a symbol is interpreted in a state.

The transition relation of a pointer program gives rise to the definition of the standard predicate transformers. The predicate transformers post (strongest postcondition) and wp (weakest precondition) are defined as usual.

5 Heap Predicates

We will abstractly represent sets of program states using formulas. We consider a logic given with respect to the signature of program variables *Var* and pointer fields *Field*. Terms in formulas are built from constant symbols $x \in \mathit{Var}$ that are interpreted as heap objects and function symbols $f \in \mathit{Field}$ that are interpreted as functions on heap objects. Formulas are interpreted in states (together with a variable assignment for the free variables). The following discussion is not restricted to a particular logic. The only further assumption we make is that the logic is closed under syntactic substitutions.

Formulas may contain free first-order variables $v_1, \ldots, v_n$. There are two equivalent ways to define the denotation of such formulas. As a running example consider the formula $\varphi(v)$ with free variable v given by:

$$\varphi(v) \equiv \; f(v) = z \;\; .$$

The intuitive way of defining the denotation $[\![\varphi(v)]\!]$ of $\varphi(v)$ is a function mapping a state s to the set of heap objects that when assigned to the free variable v satisfy φ in s:

$$\lambda\, s \in \mathit{State} \,.\, \{\, o \in \mathit{Heap} \mid s\ f\ o = s\ z \,\} \;\; .$$

For technical reasons we use an equivalent definition. Namely, we define $[\![\varphi(v)]\!]$ as a function mapping an object o to the set of all states in which φ holds if v is assigned to o:

$$[\![\varphi(v)]\!] = \lambda\, o \in \mathit{Heap} \,.\, \{\, s \in \mathit{State} \mid s\ f\ o = s\ z \,\} \;\; .$$

Syntax of expressions and commands:

$$
\begin{aligned}
x \in \mathit{Var} \; &- \; \text{set of program variables} \\
f \in \mathit{Field} \; &- \; \text{set of pointer fields}
\end{aligned}
$$

$$
\begin{aligned}
e \in \mathit{OExp} &::= x \mid e.f \\
b \in \mathit{BExp} &::= e_1 = e_2 \mid \neg b \mid b_1 \wedge b_2 \\
c \in \mathit{Com} &::= e_1 := e_2 \mid \mathsf{assume}(b)
\end{aligned}
$$

Semantics of expression and commands:

$$
\begin{aligned}
o \in \mathit{Heap} \; &- \; \text{nonempty set of allocated objects} \\
s \in \mathit{State} &\stackrel{\mathit{def}}{=} (\mathit{Var} \to \mathit{Heap}) \times (\mathit{Field} \to \mathit{Heap} \to \mathit{Heap})
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]\ s &\stackrel{\mathit{def}}{=} s\ x & [\![e_1 = e_2]\!]\ s &\stackrel{\mathit{def}}{=} [\![e_1]\!]\ s = [\![e_2]\!]\ s \\
[\![e.f]\!]\ s &\stackrel{\mathit{def}}{=} s\ f\ ([\![e]\!]\ s) & [\![\neg b]\!]\ s &\stackrel{\mathit{def}}{=} \neg([\![b]\!]\ s) \\
& & [\![b_1 \wedge b_2]\!]\ s &\stackrel{\mathit{def}}{=} [\![b_1]\!]\ s \ \wedge \ [\![b_2]\!]\ s
\end{aligned}
$$

$$
\begin{aligned}
[\![x := e]\!]\ s\ s' &\stackrel{\mathit{def}}{=} s' = s[x \mapsto [\![e]\!]\ s] \\
[\![e_1.f := e_2]\!]\ s\ s' &\stackrel{\mathit{def}}{=} s' = s[f \mapsto (s\ f)[[\![e_1]\!]\ s \mapsto [\![e_2]\!]\ s]] \\
[\![\mathsf{assume}(b)]\!]\ s\ s' &\stackrel{\mathit{def}}{=} [\![b]\!]\ s \ \wedge \ s = s'
\end{aligned}
$$

Predicate transformers:

$$
\begin{aligned}
\mathsf{post}, \mathsf{wp} \ &\in \ \mathit{Com} \to 2^{\mathit{State}} \to 2^{\mathit{State}} \\
\mathsf{post}\ \ c\ S &\stackrel{\mathit{def}}{=} \{\, s' \mid \exists s.\, [\![c]\!]\ s\ s' \ \wedge \ s \in S \,\} \\
\mathsf{wp}\ \ c\ S &\stackrel{\mathit{def}}{=} \{\, s \mid \forall s'.\, [\![c]\!]\ s\ s' \Rightarrow s' \in S \,\}
\end{aligned}
$$

Fig. 3. Syntax and semantics of pointer programs.

Definition 1 (Heap Formulas and Heap Predicates). *Let $\varphi(\overline{v})$ be a formula with n free first-order variables $\overline{v} = (v_1, \dots, v_n)$. The denotation $[\![\varphi(\overline{v})]\!]$ of $\varphi(\overline{v})$ is defined by:*

$$[\![\varphi(\overline{v})]\!] \stackrel{def}{=} \lambda\, \overline{o} \in Heap^n \,.\, \{\, s \in State \mid s, [\overline{v} \mapsto \overline{o}] \models \varphi(\overline{v}) \,\} \ .$$

We call the denotation $[\![\varphi(v)]\!]$ an n-ary heap predicate. *The set of all heap predicates is given by:*

$$HeapPred[n] \stackrel{def}{=} Heap^n \to 2^{State}.$$

We skip the parameter n for heap predicates whenever this causes no confusion. Moreover, we consider 0-ary heap predicates as state predicates and call closed heap formulas *state formulas.*

We want to implement predicate transformers (which are operations on sets of states) through operations on heap formulas. However, heap formulas denote heap predicates rather than sets of states. We now exploit the fact that for a heap predicate p we have that $p\,\overline{o}$ is a set of states. This allows us to generalize the predicate transformers from sets of states to heap predicates.

Definition 2 (Heap Predicate Transformers). *The predicate transformers* post *and* wp *are lifted to* heap predicate transformers *as follows:*

$$\begin{aligned} \mathsf{hpost}, \mathsf{hwp} \ &\in \ Com \to HeapPred \to HeapPred \\ \mathsf{hpost}\ c\, p \ &\stackrel{def}{=} \ \lambda\, \overline{o} \,.\, \mathsf{post}\ c\ (p\, \overline{o}) \\ \mathsf{hwp}\ c\, p \ &\stackrel{def}{=} \ \lambda\, \overline{o} \,.\, \mathsf{wp}\ c\ (p\, \overline{o}) \ . \end{aligned}$$

Since the heap predicate transformers are obtained from the standard predicate transformers via a simple lifting, their characteristic properties are preserved. In particular, the following proposition holds.

Proposition 1. *Let c be a command. The heap predicate transformers* hpost *and* hwp *form a Galois connection on the boolean algebra of heap predicates, i.e. for all $p, p' \in HeapPred[n]$ and $\overline{o} \in Heap^n$:*

$$\mathsf{hpost}\ c\, p\, \overline{o} \ \subseteq \ p'\, \overline{o} \iff p\, \overline{o} \ \subseteq \ \mathsf{hwp}\ c\, p'\, \overline{o} \ .$$

The operator hwp is one of the ingredients that we need to construct boolean heap programs. Therefore, it is important that it can be characterized in terms of a syntactic operation on formulas. Ideally this operation does not introduce additional quantifiers. Such a characterization of hwp exists, because the transition relation is deterministic. For the command $c = (x.f := y)$ we have e.g.:

$$\begin{aligned} \mathsf{hwp}\ c\ [\![\varphi(v)]\!] &= \lambda\, o \,.\, \{\, s \mid \forall s'.\, [\![c]\!]\ s\ s' \Rightarrow s' \in ([\![\varphi(v)]\!]\ o) \,\} \\ &= [\![\varphi(v)[f := \lambda\, v \,.\, \text{if } v = x \text{ then } y \text{ else } f(v)]]\!] \\ &= [\![v = x \wedge y = z \ \vee \ v \neq x \wedge f(v) = z]\!] \ . \end{aligned}$$

The resulting formula denotes the object in a state s whose f-successor is pointed to by z in the successor state of s under c. The correctness of the above transformation is justified by the following proposition.

Proposition 2. *Let* $\varphi(\overline{v})$ *be a heap formula. The operator* hwp *applied to the denotation of* $\varphi(\overline{v})$ *reduces to a syntactic operation:*

$$\begin{aligned}\mathsf{hwp}\ (x := e)\ [\![\varphi(\overline{v})]\!] &= [\![\varphi(\overline{v})[x := e]]\!] \\ \mathsf{hwp}\ (e_1.f := e_2)\ [\![\varphi(\overline{v})]\!] &= [\![\varphi(\overline{v})[f := \lambda\, v\, .\ \text{if}\ v = e_1\ \text{then}\ e_2\ \text{else}\ f(v)]]\!] \\ \mathsf{hwp}\ (\mathsf{assume}(b))\ [\![\varphi(\overline{v})]\!] &= [\![b\ \rightarrow\ \varphi(\overline{v})]\!]\ .\end{aligned}$$

Note that the lambda terms do not cause any problems even if we restrict to first-order logics. The function symbols that are substituted by lambda terms always occur in β-redexes, i.e. as in the example above, it is always possible to rewrite the result of the substitution to an equivalent lambda-free formula.

Due to Prop. 2 it is convenient to overload hwp both to a function on heap predicates as well as a function on heap formulas. Whenever we apply hwp to a heap formula we refer to the corresponding syntactic operation given in Prop. 2.

6 Heap Predicate Abstraction

We systematically construct an abstract post operator by following the framework of abstract interpretation [4]. Hence, we need to provide an abstract domain, as well as an abstraction and meaning function.

We propose an abstract domain that is given by a set of state formulas and is parameterized by unary heap predicates. For the rest of the paper we fix a particular finite set of unary heap predicates $\mathcal{P}$. We consider $\mathcal{P}$ to be given as a set of heap formulas with one dedicated free variable v. For notational convenience we consider $\mathcal{P}$ to be closed under negation.

Definition 3 (Boolean Heaps). *A* boolean heap *over* $\mathcal{P}$ *is a formula* Ψ *of the form:*

$$\Psi = \forall v. \bigvee_i C_i(v)$$

where each $C_i(v)$ *is a conjunction of heap predicates in* $\mathcal{P}$*. We denote the set of all boolean heaps over* $\mathcal{P}$ *by* $\mathit{BoolHeap}$*.*

In order to allow our analysis to treat joins in the control flow adequately, we take the disjunctive completion over boolean heaps as our abstract domain.

Definition 4 (Abstract Domain). *The* abstract domain *over* $\mathcal{P}$ *is the pair* $\langle \mathit{AbsDom}, \models \rangle$*, where* AbsDom *is given by all disjunctions of boolean heaps:*

$$\mathit{AbsDom} \stackrel{\mathit{def}}{=} \{ \bigvee_{\Psi \in F} \Psi \mid F \subseteq_{\mathit{fin}} \mathit{BoolHeap} \}\ .$$

The partial order $\models$ *on elements in* AbsDom *is the entailment relation on formulas.*

A boolean heap can be represented as a set of bitvectors over $\mathcal{P}$, one bitvector for each conjunction. Hence, it is easy to see that the abstract domain is isomorphic to sets of sets of bitvectors over $\mathcal{P}$. Moreover, the abstract domain is finite and both closed under disjunctions and conjunctions[2]. Therefore, it forms a complete lattice.

The meaning function γ that maps elements of the abstract domain to sets of states is naturally given by the denotation function, i.e. each formula Ψ of the abstract domain is mapped to the set of its models:

$$\begin{aligned} \gamma &\in \mathit{AbsDom} \to 2^{\mathit{State}} \\ \gamma\, \Psi &\stackrel{def}{=} [\![\Psi]\!] \ . \end{aligned}$$

The abstraction function α is determined by:

$$\begin{aligned} \alpha &\in 2^{\mathit{State}} \to \mathit{AbsDom} \\ \alpha\, S &\stackrel{def}{=} \bigwedge \{\, \Psi \in \mathit{AbsDom} \mid S \subseteq [\![\Psi]\!] \,\} \ . \end{aligned}$$

The function γ distributes over conjunctions and is thus a complete meet morphism. Together with the fact that we defined α as the best abstraction function with respect to γ, we can conclude that α and γ form a Galois connection between concrete and abstract domain:

$$\langle 2^{\mathit{State}}, \subseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \mathit{AbsDom}, \models \rangle \ .$$

If we consider a state s, the abstraction function α maps the singleton $\{s\}$ to the smallest boolean heap that is valid in s. This boolean heap describes the boolean covering of heap objects with respect to the heap predicates $\mathcal{P}$.

In order to describe these smallest boolean coverings, we assign an *abstract object* $\alpha_s(o)$ to every object o and state s. This abstract object is given by a monomial (complete conjunction) of heap predicates and represents the equivalence class of all objects that satisfy the same heap predicates as o in s:

$$\alpha_s(o) \stackrel{def}{=} \bigwedge \{\, p(v) \in \mathcal{P} \mid s, [v \mapsto o] \models p(v) \,\} \ .$$

The smallest boolean heap that abstracts s consists of all abstract objects $\alpha_s(o)$ for objects $o \in \mathit{Heap}$. Formally, the abstraction of a set of states S is characterized as follows:

$$\alpha\, S \equiv \bigvee_{s \in S} \forall v.\ \bigvee_{o \in \mathit{Heap}} \alpha_s(o) \ .$$

7 Cartesian Abstraction

According to [4] the best abstract post operator $\mathsf{post}^{\#}$ is given by the composition of α, post and γ. In the following, we fix a command c and consider all

[2] Conjunctions distribute over the universal quantifiers in boolean heaps.

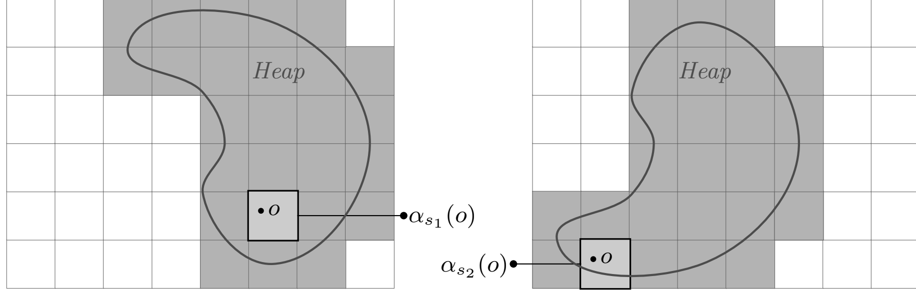

Fig. 4. The boolean heaps for two states s_1 and s_2. The same object $o \in \mathit{Heap}$ falls into different equivalence classes $\alpha_{s_1}(o)$ and $\alpha_{s_2}(o)$ for each of the states s_1 and s_2. This leads to a different boolean covering of the set *Heap* in the two states and hence to different boolean heaps.

applications of predicate transformers with respect to this particular command. Using the characterization of α from the previous section, we get:

$$\mathsf{post}^\#(\Psi) \stackrel{def}{=} \alpha \circ \mathsf{post} \circ \gamma(\Psi) \equiv \bigvee_{s \in [\![\Psi]\!]} \forall v. \bigvee_{o \in \mathit{Heap}} \alpha_{\mathsf{post}(\{s\})}(o) \ .$$

In order to compute the image of Ψ under $\mathsf{post}^\#$ we need to check for each boolean heap whether it appears as one of the disjuncts in $\mathsf{post}^\#(\Psi)$. Given that n is the number of (positive) heap predicates in $\mathcal{P}$, considering all 2^{2^n} boolean heaps explicitly results in a doubly-exponential running time for the computation of $\mathsf{post}^\#$. Therefore, our goal is to develop an approximation of the best abstract post operator that can be easily implemented. However, we require this operator to be formally characterized in terms of an abstraction of $\mathsf{post}^\#$.

Since the best abstract post operator distributes over disjunctions, we characterize the abstraction of $\mathsf{post}^\#$ on boolean heaps rather than their disjunctions. In the following, consider the boolean heap Ψ given by:

$$\Psi = \forall v. \psi(v) \ .$$

As illustrated in Fig. 5, the problem is that even if we apply $\mathsf{post}^\#$ to the single boolean heap Ψ, its image under $\mathsf{post}^\#$ will in general be a disjunction of boolean heaps. We first abstract a disjunction of boolean heaps by a single boolean heap. This is accomplished by merging all coverings represented by boolean heaps in $\mathsf{post}^\#(\Psi)$ into a single one. That means the resulting single boolean heap represents a covering of all objects for all states that are models of $\mathsf{post}^\#(\Psi)$.

We define the best abstractions of the heap predicate transformers with respect to the set of all boolean combinations of heap predicates in $\mathcal{P}$ (denoted by $\mathcal{BC}(\mathcal{P})$):

$$\mathsf{hpost}^\#(\psi(v)) \stackrel{def}{=} \bigwedge \{ \varphi(v) \in \mathcal{BC}(\mathcal{P}) \mid \psi(v) \models \mathsf{hwp}(\varphi(v)) \}$$

$$\mathsf{hwp}^\#(\psi(v)) \stackrel{def}{=} \bigvee \{ \varphi(v) \in \mathcal{BC}(\mathcal{P}) \mid \varphi(v) \models \mathsf{hwp}(\psi(v)) \} \ .$$

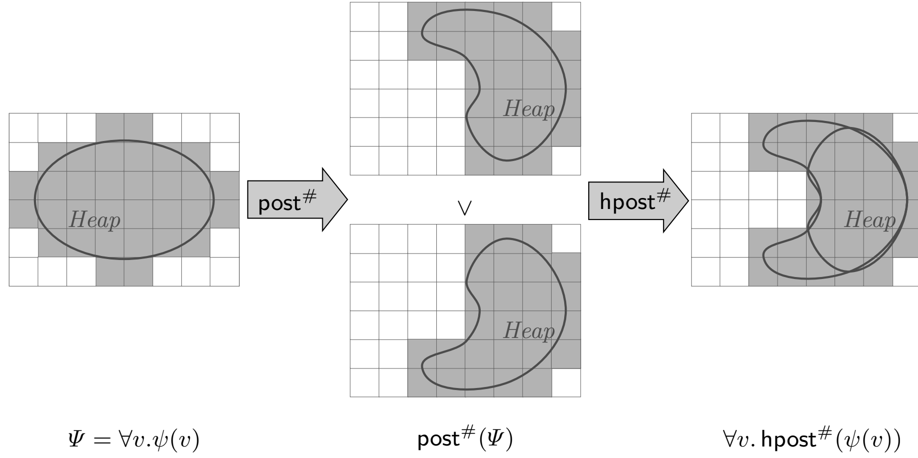

Fig. 5. Application of $\mathsf{post}^\#$ to a single boolean heap Ψ and its approximation using $\mathsf{hpost}^\#$.

By Prop. 1 and the definition of $\mathsf{hpost}^\#$ respectively $\mathsf{hwp}^\#$ it is easy to see that these two operators again form a Galois connection on the set of all boolean combinations of heap predicates in $\mathcal{P}$.

Formally, the approximation of the best abstract post that we described above corresponds to the application of the best abstraction of the operator hpost to the heap formula $\psi(v)$.

Proposition 3. *Let $\Psi = \forall v.\psi(v)$ be a boolean heap. Applying $\mathsf{hpost}^\#$ to $\psi(v)$ results in an abstraction of $\mathsf{post}^\#(\Psi)$:*

$$\mathsf{post}^\#(\Psi) \models \forall v.\, \mathsf{hpost}^\#(\psi(v)) \enspace .$$

Since the operator $\mathsf{hpost}^\#$ again distributes over disjunctions, we can compute the new covering by applying $\mathsf{hpost}^\#$ *locally* to each disjunct in $\psi(v)$. That is, if $\psi(v)$ is given as a disjunction of abstract objects:

$$\psi(v) = \bigvee_i C_i(v)$$

then for each $C_i(v)$ we compute the new covering $\mathsf{hpost}^\#(C_i(v))$ of objects represented by $C_i(v)$ for the states satisfying $\mathsf{post}^\#(\Psi)$.

However, computing this localized post operator is still an expensive operation. The result of $\mathsf{hpost}^\#$ applied to an abstract object will in general be a disjunction of abstract objects. We face the same problem as before: we would have to consider all 2^n monomials over heap predicates, in order to compute the precise image of a single abstract object under $\mathsf{hpost}^\#$.

A disjunction of conjunctions over abstraction predicates can be represented as a set of bitvectors. In the context of predicate abstraction one uses Cartesian

abstraction to approximate sets of bitvectors [3]. For a set of bitvectors represented by a boolean formula $\psi(v)$, the Cartesian abstraction $\alpha_{\mathsf{Cart}}(\psi(v))$ is given by the smallest conjunction over abstraction predicates that is implied by $\psi(v)$:

$$\alpha_{\mathsf{Cart}}(\psi(v)) \stackrel{def}{=} \bigwedge \{\, p(v) \in \mathcal{P} \mid \psi(v) \models p(v) \,\} \ .$$

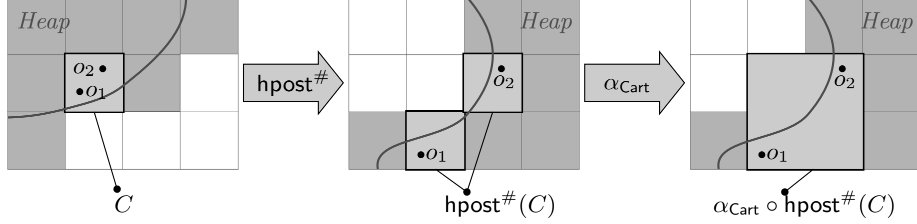

Fig. 6. Application of $\mathsf{hpost}^{\#}$ to a single abstract object C and the approximation under α_{Cart}.

Figure 6 sketches the idea of Cartesian abstraction in our context. It abstracts all abstract objects in the image under the operator $\mathsf{hpost}^{\#}$ by a single conjunction. Composing the operator $\mathsf{hpost}^{\#}$ with the Cartesian abstraction function gives us our final abstraction of the best abstract post operator.

Definition 5 (Cartesian Post). *Let $\Psi = \forall v. \bigvee_i C_i(v)$ be a boolean heap. The* Cartesian post *of Ψ is defined as follows:*

$$\mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) \stackrel{def}{=} \forall v. \bigvee_i \alpha_{\mathsf{Cart}} \circ \mathsf{hpost}^{\#}(C_i(v)) \ .$$

We extend the Cartesian post to a function on AbsDom in the natural way by pushing it over disjunctions of boolean heaps.

Theorem 1 (Soundness of Cartesian Post). *The Cartesian post is an abstraction of $\mathsf{post}^{\#}$:*

$$\forall \Psi \in \mathit{BoolHeap}.\ \mathsf{post}^{\#}(\Psi) \models \mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) \ .$$

Proof. Let $\Psi = \forall v. \bigvee_i C_i(v)$ be a boolean heap. The statement follows immediately from Prop. 3 and the fact that for every $C_i(v)$ we have $C_i(v) \models \alpha_{\mathsf{Cart}}(C_i(v))$.

Theorem 2 (Characterization of Cartesian Post). *Let $\Psi = \forall v. \bigvee_i C_i(v)$ be a boolean heap. The Cartesian post of Ψ is characterized as follows:*

$$\mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) \equiv \forall v. \bigvee_i \bigwedge \{\, p(v) \in \mathcal{P} \mid C_i(v) \models \mathsf{hwp}^{\#}(p(v)) \,\} \ .$$

Proof. Using the fact that $\mathsf{hpost}^{\#}$ and $\mathsf{hwp}^{\#}$ form a Galois connection $(*)$ we have:

$$\begin{aligned}
\mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) &\equiv \forall v. \bigvee_i \alpha_{\mathsf{Cart}} \circ \mathsf{hpost}^{\#}(C_i(v)) && \text{Def. of } \mathsf{post}^{\#}_{\mathsf{Cart}} \\
&\equiv \forall v. \bigvee_i \bigwedge \{\, p(v) \in \mathcal{P} \mid \mathsf{hpost}^{\#}(C_i(v)) \models p(v) \,\} && \text{Def. of } \alpha_{\mathsf{Cart}} \\
&\equiv \forall v. \bigvee_i \bigwedge \{\, p(v) \in \mathcal{P} \mid C_i(v) \models \mathsf{hwp}^{\#}(p(v)) \,\} && \text{by } (*) \ .
\end{aligned}$$

Summarizing the above result, the image of a boolean heap Ψ under $\mathsf{post}^{\#}_{\mathsf{Cart}}$ is constructed by updating for each monomial C_i in Ψ the values of all heap predicates in C_i. These updates are computed by checking for each heap predicate p whether C_i implies the weakest precondition of p or its negation. Hence, the Cartesian post operator $\mathsf{post}^{\#}_{\mathsf{Cart}}$ corresponds to a boolean heap program as it is defined in Fig. 2. The crucial part in the construction of the boolean heap program lies in the computation of the operator $\mathsf{hwp}^{\#}$. It is implemented using a syntactic operation on formulas (the operator hwp) and by calls to a theorem prover for the entailment tests.

Discussion. Already the first abstraction of the best abstract post operator that we gave above can be formally characterized in terms of a Cartesian abstraction. This leads to a slightly more precise abstraction of $\mathsf{post}^{\#}$; see [14] for details. However, this abstraction is more expensive and introduces a dependency of the operator $\mathsf{hwp}^{\#}$ on the abstract state Ψ for which we compute the post. This dependency violates our goal to have a decoupling of the abstraction phase from the fixed point computation of the analysis.

Focus and Coerce. Cartesian abstraction does not introduce an additional loss of precision as long as the abstract system behaves deterministically, i.e. every abstract object is mapped again to a single abstract object under the operator $\mathsf{hpost}^{\#}$. However, for some commands, e.g. when one iterates over a recursive data structure, the abstract system will behave inherently nondeterministically. In some cases the loss of precision that is caused by this nondeterminism cannot be tolerated. A similar problem occurs in the context of three-valued shape analysis. In [13] so called *focus* and *coerce* operations are used to solve this problem. These operations split three-valued logical structures according to weakest preconditions of predicates and thereby handle the nondeterminism in the abstraction.

Though the *focus* and *coerce* operations are conceptually difficult, it is possible to define a simple corresponding splitting operation in our framework. This operation can be explained in terms of a temporary refinement of the abstract domain. Namely, for splitting one first refines the abstraction by adding new abstraction predicates given by the weakest preconditions of the abstraction predicates that cause the nondeterminism. The refinement causes a splitting of abstract objects and boolean heaps, such that each abstract object in each

boolean heap has precise information regarding the weakest preconditions of the problematic predicates. This guarantees that the Cartesian post computes precise updates for these predicates. After computing the Cartesian post the result is mapped back to the original abstract domain by removing the previously added predicates. For a more detailed discussion again see [14].

8 Conclusion

We showed how the abstraction originally proposed in three-valued shape analysis can be constructed in the framework of predicate abstraction. The consequences of our results are:

- a different view on the underlying concepts of three-valued shape analysis.
- a framework of *symbolic* shape analysis. Symbolic means that the abstract post operator is an operation over formulas and is itself constructed solely by deductive reasoning.
- a clear phase separation between the computation of the abstraction and the computation of the fixed point. Among other potential advantages this allows the offline computation of the abstract post operator.
- the possibility to use efficient symbolic methods such as BDDs or SAT solvers. In particular, the abstract post operator itself can be represented as a BDD.

Our framework does not *a priori* impose any restrictions on the data structures implemented by the analyzed programs. Such restrictions only depend on the capabilities of the underlying theorem prover which is used for the entailment tests. There is ongoing research on how to adapt or extend existing theorem provers and decision procedures to the theories that are needed in the context of shape analysis; see e.g. [7, 8]. This is a challenging branch for further research.

Another direction for future work is to study refinements of our abstract domain that are even closer to the abstract domain used in three-valued shape analysis. Evidently our framework extends from unary heap predicates to heap predicates of arbitrary arity. If we allow binary relations over heap objects in the abstract domain, we obtain the universal fragment of shape analysis constraints [10].

Extending the abstract domain to the full boolean algebra of shape analysis constraints and thus having exactly the same precision as using three-valued logical structures is more involved. In that case we allow both universally and existentially quantified boolean combinations of heap predicates as base formulas for our abstract domain. Normal forms of conjunctions of these base formulas are an extension of boolean heaps. They correspond to possible boolean coverings of objects with *nonempty* monomials over heap predicates. These normal forms can again be represented as sets of bitvectors over heap predicates. However, in this case it is not clear how Cartesian abstraction can be applied in a way that is similar to the case where we restrict to the universal fragment.

Acknowledgments. We thank Viktor Kuncak, Patrick Lam, and Alexander Malkis for comments and suggestions.

References

1. I. Balaban, A. Pnueli, and L. Zuck. Shape Analysis by Predicate Abstraction. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, LNCS 3385, pages 164–180. Springer, 2005.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Programming language design and implementation (PLDI'01)*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213, 2001.
3. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pages 268–283. Springer, 2001.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, 1979.
5. D. Dams and K. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, LNCS 2575, pages 310–323. Springer, 2003.
6. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification (CAV'97)*, LNCS 1254, pages 72–83. Springer, 1997.
7. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The Boundary Between Decidability and Undecidability for Transitive-Closure Logics. In *Computer Science Logic (CSL 2004)*, LNCS 3210, pages 160–174. Springer, 2004.
8. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification Via Structure Simulation. In *Computer Aided Verification (CAV'04)*, LNCS 3114, pages 281–294. Springer, 2004.
9. N. Klarlund and M. Schwartzbach. Graph types. In *Symposium on Principles of Programming Languages (POPL'93)*, pages 196–205, 1993.
10. V. Kuncak and M. Rinard. Boolean Algebra of Shape Analysis Constraints. In *Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, LNCS 2937, pages 59–72. Springer, 2004.
11. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, LNCS 3385, pages 181–198. Springer, 2005.
12. A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Programming language design and implementation (PLDI'01)*, pages 221–231, 2001.
13. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
14. T. Wies. Symbolic Shape Analysis. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 2004.
15. G. Yorsh. Logical Characterizations of Heap Abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003.
16. G. Yorsh, T. Reps, and M. Sagiv. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS 2988, pages 530–545. Springer, 2004.