Separating Shape Graphs

Vincent Laviron[1], Bor-Yuh Evan Chang[2], and Xavier Rival[1,3]

[1] École Normale Supérieure, Paris, France
[2] University of Colorado, Boulder, Colorado, USA
[3] INRIA Rocquencourt, France
laviron@di.ens.fr, bec@cs.colorado.edu, rival@di.ens.fr

Abstract. Detailed memory models that expose individual fields are necessary to precisely analyze code that makes use of low-level aspects such as, pointers to fields and untagged unions. Yet, higher-level representations that collect fields into records are often used because they are typically more convenient and efficient in modeling the program heap. In this paper, we present a shape graph representation of memory that exposes individual fields while largely retaining the convenience of an object-level model. This representation has a close connection to particular kinds of formulas in separation logic. Then, with this representation, we show how to extend the Xisa shape analyzer for low-level aspects, including pointers to fields, C-style nested structures and unions, malloc and free, and array values, with minimal changes to the core algorithms (e.g., materialization and summarization).

1 Introduction

At the core of precise program analyzers, such as verification tools and shape analyses, is an abstract memory model that represents the program heap. The design of such representations for C code is particularly challenging because of a tension between keeping it simple and supporting low-level pointer manipulation. Specifically, the level of detail exposed in an abstract memory model determines whether the analyzer can even reason about particular low-level aspects. For example, does the representation allow the addressing expression `&(p->f1.f2)` (i.e., taking the address of a nested structure or union field), while supporting basic field read expressions `p->f` easily?

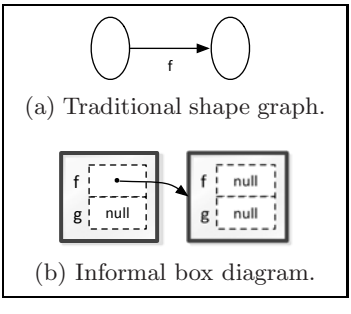


Fig. 1. A shape graph for Java-like structures

To illustrate this tension, we show in Fig. 1, a simple, traditional shape graph (a) that represents the concrete memory shown in (b) as an informal box diagram. In this shape graph, each node corresponds to an object (i.e., a record of fields) and each edge stands for a points-to relation between objects. Historically, shape analyzers have focused on Java-like structures where memory can

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 387–406, 2010.

be viewed as a simple collection of objects with field reads but no complex addressing expressions (e.g., [23,24,11]). Consequently, such a shape graph is convenient and widely used. For example, TVLA [24] uses three-valued logical structures to encode such shape graphs. In TVLA, the shape graph in Fig. 1a corresponds to the two-valued formula $\mathsf{f}(u_1, u_2)$ over individuals u_1 and u_2. Similarly, in separation logic [22] and separation logic-based shape analyzers, we might represent the memory shown in Fig. 1b with the following formula: $u_1 \mapsto \{\mathsf{f}\colon u_2, \mathsf{g}\colon \mathsf{null}\} \ * \ u_2 \mapsto \{\mathsf{f}\colon \mathsf{null}, \mathsf{g}\colon \mathsf{null}\}$ as, for example, in Berdine *et al.* [1]. This formula says that there are two disjoint records pointed to by u_1 and u_2, each with fields `f` and `g`, and where the `f` field of u_1 points to u_2 (and all other fields are `null`).

We see that something more detailed is needed to express, for example, a pointer to a field (i.e., `&(p->f)`). In particular, we must expose the individual fields (i.e., the components of a record) as shown informally in Fig. 2. However, if we simply take the components as the unit memory cells, then we lose the object-level structure. Such an object-level view is convenient for the common case with Java-like structures and necessary for the sound modeling of object-level properties, such as for analyzing uses of malloc and free.

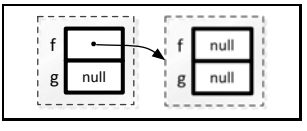


Fig. 2. Box diagram with exposed components

In this paper, we make the following contributions:

- We describe a graph-based memory representation that permits reasoning at the object level and the field level simultaneously. The key idea is to represent abstract memory cells with edges (rather than nodes as is traditional) and to view nodes as aggregates of their outgoing edges. This simple shift in view allows us to cleanly separate object-level properties from field-level ones. In particular, we show how this representation can be instantiated in different ways to express varying degrees of detail (Sect. 3).
- We present a particular instantiation of separating shape graphs to model low-level aspects of C, including pointer-to-field (i.e., `&(p->f)`), nested structures, untagged unions, C-style malloc-free, and array values (Sect. 4).
- We demonstrate the applicability of our representation by extending the Xisa analyzer [6,5] for these low-level aspects of C, often unhandled in shape analyzers. In particular, our extension required minimal changes to the original object-based algorithms for materialization and summarization that are key to shape analysis (Sect. 5).

To motivate and provide intuition for separating shape graphs, the next section (Sect. 2) presents an example shape analysis with nested structures and unions.

2 Background and Overview

In separation logic, the record-level points-to relation is a standard abbreviation for separated points-to relations of components:

$$e \mapsto \{f_1\colon e_1, \ldots, f_n\colon e_n\} \quad \stackrel{\text{def}}{=} \quad e \,@f_1 \mapsto e_1 * \cdots * e \,@f_n \mapsto e_n \qquad (\star)$$

where e @f is a field offset expression (i.e., the offset corresponding to field f from the base pointer given by expression e). By using formulas of the form on the right side, we essentially expose individual fields (referred by Parkinson as taking a field-splitting model of memory [21]).

For shape analysis of C code, we essentially want a representation that minimizes the need to convert back-and-forth between the left and right-hand sides of definition ($\star$). In other words, we want a model that exposes individual fields and permits complex addressing expressions but maintains object-level structure. To begin, consider the graph shown in Fig. 3. A node denotes a value (e.g., a memory address, an integer, null) labeled by a *symbolic value*, which is an existentially quantified variable. We use lowercase Greek letters ($\alpha, \beta, \gamma, \ldots$) to range over symbolic values. An edge corresponds to a unit memory cell or a points-to relation at the component level (like on the right in definition ($\star$) and Fig. 2); for instance, the edge between α and β says that at field f from address α, the contents of the cell is β. At the same time, if we ensure that all symbolic values correspond to base pointers of objects, we have a representation that also can be read as an object-level formula (like on the left in definition ($\star$) and Fig. 1b) and looks fairly similar to the traditional shape graph in Fig. 1a. It is important to note that edges represent disjoint memory cells (like nodes in traditional shape graphs) but that nodes may correspond to the same concrete value (unlike in traditional shape graphs).

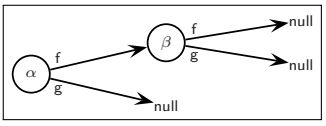
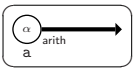


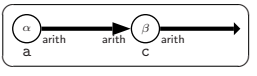
Fig. 3. A simple separating shape graph for the concrete memory in Fig. 1b

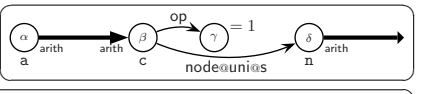
Thus far, we have a shape graph representation for Java-like structures as in our prior work [6,5] and similarly used by others (e.g., [1]). In this paper, we make the connection between such graphs and a restricted language of separation logic formulas explicit and take this view further to capture low-level aspects of C (see Sect. 3). In the remainder of this section, we provide intuition for our C-level model of memory with an example shown in Fig. 4. In particular, we consider the analysis of code for evaluating arithmetic expressions represented by a syntax tree (using the C-type `Arith`). These syntax trees feature several kinds of nodes (constants `cst`, unary operators `uni`, and binary operators `bin`) that are encoded with a union type. The `op` field is a tag, or discriminant, that indicates which branch of the union field `node` is being used.

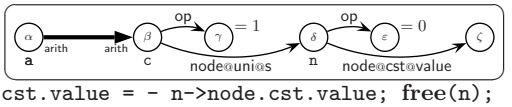
Our shape analysis proceeds by abstract interpretation [10] computing sound local invariants at each program point (i.e., graphs that over-approximate the set of possible concrete memory states). In the figure, we show the local invariants inferred by our analysis boxed and right-justified at a number of program points. Like in our prior work [6,5], a thick edge, or a *checker edge*, represents a memory region summary (i.e., a set of points-to edges abstractly). Our abstract domain is parametric in inductive definitions that give rise to such summaries. These user-supplied inductive definitions come in the form of *invariant checkers*; that is, they can be viewed as code that traverses a data structure to check a run-time

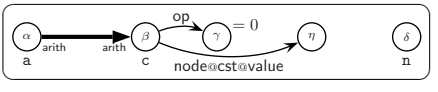
```
typedef struct Arith {
  char op; // the tag: 0 for constants, 1 for unary −, 2 for +, 3 for ∗, ...
  union {
    struct { long long int value; } cst; // a constant value
    struct { struct Arith* s; } uni; // a unary operator
    struct { struct Arith* l; struct Arith* r; } bin; // a binary operator
  } node;
} Arith;

1  int eval(Arith* a) {
2
     Arith* c; Arith* n;
3    while (a->op != 0) { // 0 stands for a constant
4      c = a; // initialize the cursor
5      while (1) { // traverse a branch until some
                   // simplification can be done
6
         if (c->op == 1) {
7          // 1 stands for unary negation
           n = c->node.uni.s;
8
           if (n->op == 0) {
9            // simplify
             c->op = 0; c->node.cst.value = - n->node.cst.value; free(n);
10
             break;
           }
11         else { c = n; }
12       }
13       else { ... /* other cases for addition, multiplication, etc. */ ... }
14     }
15   }
16
17   return a->node.cst.value;
18 }
```











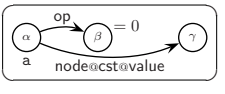


Fig. 4. An example analysis of a syntax tree operation using C-style unions

invariant. The first invariant at program point 2 indicates the pre-condition that a is a well-formed syntax tree given by the inductive checker arith. For now, we focus on the analysis (the formal definition of the arith checker will be given in Sect. 4.2). The loop invariant in the second loop at program point 6 expresses the fact that c points to a subtree of the syntax tree pointed to by a; specifically, there is a syntax tree *segment* between α and β described by a partial instance of checker arith, and separately, there is a completion of the tree from β.

The condition test on field op of c causes the analyzer to *unfold* the definition of arith for the syntax tree pointed to by c to produce the invariant at program

point 8. Unfolding is a partial concretization or refinement step that *materializes* points-to edges, typically by considering cases. Here, the fact that $\gamma = 1$ (given by the guard that `c->op == 1`) leaves the one case where the `uni` branch of the union is active. Thus, we have two points-to edges materialized: one labeled with `op` for the `op` field of `c` and one labeled with `node@uni@s` for the subexpression pointer of the unary operator (i.e., `c->node.uni.s`). Data constraints, such as $\gamma = 1$, are captured by a *base abstract domain* that is also a parameter of our analysis; in our examples, we will note such constraints as necessary and assume that we are using a base domain that can capture them. Next, a similar unfolding happens at the syntax tree node pointed to by `n` (i.e., δ) to produce the invariant at program point 9. At this point, the structure below `c` is fully materialized, and the subsequent sequence of updates is reflected by modifications and deletions of points-to edges to produce the invariant at program point 10. The result of the outer evaluation loop is a syntax tree node for a constant corresponding to the inferred invariant at program point 17.

As alluded to earlier, the key challenge addressed in this paper is creating an analysis that is capable of reasoning about low-level C features, such as unions, while not unnecessarily complicating the analysis of higher-level, Java-like code. This challenge is highlighted in this example analysis. Focus again on the transition between points 9 and 10. On one hand, the view of the syntax tree node pointed to by `c` changes by writing through `node.cst.value` in the second statement, which is not evident in the state at point 9. Analyzing code with unions requires careful management of several such *views* of the same or overlapping memory cells (e.g., `c->node`) (Sect. 3.2); such views are accompanied by fields of varying *sizes* and thus necessitating delicate treatment of values and memory cells (Sect. 4). At the same time, the first statement updates the `op` tag, which is an ordinary field except for its role in discriminating the union. We want the modeling of this field to be largely independent of the complexities introduced by the union type (Sect. 3.1).

The most intricate aspect of most program analysis algorithms is the widening operator that extrapolates loop invariants—in our domain, it folds points-to edges into checker edges. Our algorithm is no exception but by encapsulating unions within a shape graph representation, our widening operator described in great detail in earlier papers [6,5] remains essentially unchanged (Sect. 4.3)

3 Separating Shape Graphs for Modeling Memory

In this section, we gradually evolve an abstract domain for shape analysis to model successively lower-level aspects. In particular, we want a domain that can be instantiated differently depending on the desired level of detail. For example, we may want to analyze code that relies on compiler implementation-specific details such as the size and packing of fields, or we may want to be compiler independent and reject certain low-level idioms. In Sect. 4, we formalize a particular instantiation with the lower-level aspects.

In Fig. 5, we show such an abstract domain using separating shape graphs. Separating shape graphs represent memories M, which consist of either an empty

memories	M	$::=$	emp	empty
		$\mid$	$M_1 * M_2$	separate regions
		$\mid$	S	region summary
		$\mid$	$\alpha\ \ell \overset{q}{\mapsto} \beta\ r$	memory cell
summaries	S			
l-exprs, r-exprs	ℓ, r			
points-to properties	q			
variable environment	E	$::=$	$\cdot \mid E, x \mapsto \alpha$	static scope
value constraints	P	$\in$	$\mathbb{P}^\sharp$	base domain
analysis state	A	$::=$	$\bot \mid \exists \vec{\alpha}.\, \langle E, M, P\rangle \mid A_1 \vee A_2$	disjunctive domain

Fig. 5. An abstract shape domain with separating shape graphs (M)

memory emp or separate regions $M_1 * M_2$ like in separation logic. Regions may be either a summary region S or a points-to relation. Summaries S abstract some configuration of points-to edges. They are necessary to capture an unbounded number of configurations needed by shape analysis. For example, in Xisa [6,5], summaries consist of instances of inductive definitions and inductive segments derived from user-supplied checkers. However, because their exact form is unimportant here, we leave them unspecified. Instead, in this section, we focus on *fully unfolded separating shape graphs*, or *unfolded graphs* for short, that consist only of points-to edges.[1]

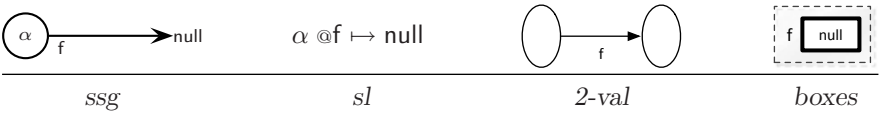
The rest of the analysis state is straightforward and mostly standard. We have an environment E that maps variables to their *addresses*, which allows us to model address-of-locals (i.e., `&x`), as in our prior work [6,5]. A base abstract domain $\mathbb{P}^\sharp$ tracks constraints on values (e.g., α, β). Overall, an analysis state is a finite disjunction of $\langle E, M, P\rangle$ tuples; we simply make explicit that symbolic values are existential variables at the analysis state-level. Recall from Sect. 2 that nodes correspond to values—typically base pointers of objects. The points-to relation $\alpha\ \ell \overset{q}{\mapsto} \beta\ r$ is an edge from α to β and represents a singleton memory cell (e.g., fields in the case of a field-splitting model for Java-like structures). The address expression ℓ and contents expression r allow for computing offsets from base values α and β, respectively. Finally, we allow edges to be decorated with properties q, which are used in the subsequent subsections. Note how memory layout properties (e.g., field size) can be captured on edges, while value properties (e.g., type of a value, range of an integer constant) refer to nodes. In particular, memory cells are modeled by *edges* not nodes as with traditional shape graphs.

To obtain separating shape graphs for Java-like structures (as in our prior work [6,5]), we simply define ℓ and r as shown inset. That is, we allow field offsets $@f$ on the left but only base pointers on the right (ε indicates empty). Pictorially, we show an example separating shape graph (*ssg*) along with the corresponding separation logic formula (*sl*), two-valued structure (*2-val*), and informal box diagram (*boxes*) in the following:

$$\ell ::= \varepsilon \mid @f$$
$$r ::= \varepsilon$$

[1] Unfolded graphs are analogous to TVLA's two-valued structures.

Example 1 (A Java-Like Structure with One Field).



Kreiker *et al.* in a recent paper [18] present a "fine-grained semantics" in the context of TVLA using one node per component plus one node for the enclosing record in order to model pointer-to-field and nested structures. In Example 1, the two-valued structures we draw uses this one node per component model.[2] While the picture looks the same as Fig. 1a, the two-valued structure above represents an object with one field f not two objects.

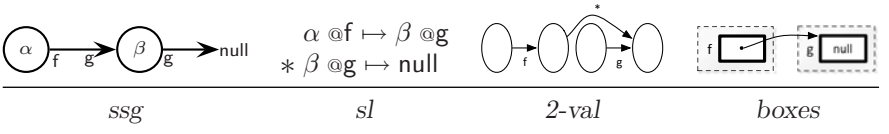
In the subsequent sections, we consider low-level aspects of the C memory model. Note that we do not directly jump to a byte-level or assembly-level model, rather we want object-level notions like field names to coexist with lower-level aspects like numeric offsets. We first consider lowerings that are mostly compiler independent based solely on the semantics of C (Sect. 3.1) followed by those that are compiler dependent, such as field sizes (Sect. 3.2).

3.1 Compiler Independent Use of Aggregates

Internal Pointers. With individual fields exposed, the extension to internal pointers becomes clear. We simply need to allow field offset expressions on the right side of points-to in addition to the left side, instantiating ℓ and r as shown in the inset.

$$\ell ::= \varepsilon \mid @f$$
$$r ::= \ell$$

Example 2 (An Internal Pointer). A pointer to the g field of a structure is represented as follows:



Nested Structures. The base pointer of a nested structure is the field offset of its enclosing struct, so we need to allow for a path of field offsets with ℓ and r as defined in the inset. With nested structures, the contents or value of a field may be a record (i.e., another structure of subfields). Thus, in our representation, symbolic values may now take on record values $\{f_1 \colon \beta_1\ r_1, \ldots, f_n \colon \beta_n\ r_n\}$ in addition to, for example, null and integers. As a consequence, we may need to

$$\ell ::= \varepsilon \mid \ell @ f$$
$$r ::= \ell$$

[2] Note that in that paper, they also present a "coarse-grained semantics" that goes back to the one node per record model while retaining the ability to reference field pointers and nested structures. In addition to pointer-to-field and nested structures, we also consider untagged unions, field sizes, and array values in this paper.

$$\textsc{Var}\ \frac{(E(x) \mapsto \beta\ r) \in M}{x \downarrow (E(x) \mapsto \beta\ r)} \qquad \textsc{Deref}\ \frac{e \Downarrow \beta\ r_1}{*e \downarrow (\beta\ r_1 \mapsto \gamma\ r_2)} \qquad \textsc{FieldDeref}\ \frac{e \Downarrow \beta\ r_1}{e{\rightarrow}f \downarrow (\beta\ r_1 @ f \mapsto \gamma\ r_2)}$$

$$\textsc{RVal}\ \frac{e \downarrow (\alpha\ \ell \mapsto \beta\ r)}{e \Downarrow \beta\ r} \qquad \textsc{AddrOf}\ \frac{e \downarrow (\alpha\ \ell \mapsto \beta\ r)}{\&e \Downarrow \alpha\ \ell} \qquad \textsc{FieldOffset}\ \frac{e \downarrow (\alpha\ \ell \mapsto \beta\ r_1)}{e.f \downarrow (\alpha\ \ell @ f \mapsto \gamma\ r_2)}$$

Fig. 6. Evaluation of program addressing expressions for structures

reduce between an edge containing a record and a set of edges of its components (essentially using definition ($\star$) in Sect. 2).

Example 3 (Reduction for Nested Structures). Consider the following nested structure declaration: **typedef struct { struct { int i; int j; } t; } S;**. Then, the following is an example equivalence:

Thus far, we have been able to view all points-to edges as word-sized cells containing word-sized values. We now have irregularly-sized cells and values and thus want to ensure that updates at least respect cell size. We consider this issue further in Sect. 3.2.

Program Expression Evaluation. In an unfolded graph, pointer updates amount to the swinging of an edge. Such a destructive update is sound because of separation. To determine which edge to swing and how to update it, we traverse the graph starting from variables and following dereferences to find the edges corresponding the cell being written and the cells being read. To describe this traversal precisely, we define two judgments $e \downarrow_{E,M} (\alpha\ \ell \mapsto \beta\ r)$ and $e \Downarrow_{E,M} \beta\ r$ that evaluate a program expression e in an environment E and graph M to yield a cell and a value, respectively, in Fig. 6. These expressions allow dereferences of internal pointers, access of arbitrarily nested structures, and taking the address of any cell. To keep the rules concise, we elide the environment and graph parameters, as they are constant. Furthermore, we implicitly require that any edge appearing in the rules exists in the graph (though we show this side-condition explicitly in Var as an example). The cell of a variable x is the one whose left side of points-to is the address given by the environment (Var). Dereferences $*e$ and $e{\rightarrow}f$ follow an edge, that is, they get the value of their subexpression and find the edge whose left side of points-to is that value or that value plus the field offset, respectively (Deref and FieldDeref). To find the value of an expression, we simply find the cell corresponding to the expression and yield the right side (i.e., the contents) (RVal). The rule for the address-of operator $\&e$ (AddrOf) is more interesting in that its role as converting l-values into r-values is made evident. In particular, it makes sure the cell corresponding to e exists and then

returns its address. Thus, l-expressions ℓ must be contained in r-expressions r to capture internal pointers, as addresses can be returned as values. The rule for the field offset expression $e.f$ (FIELDOFFSET) captures the shift from a points-to edge representing a record to edges for its components (see Example 3). Note that the cell evaluation judgment $e \downarrow (\alpha\ \ell \mapsto \beta\ r)$ is needed only for expressions $\&e$ and $e.f$ (i.e., the value evaluation judgment $e \Downarrow \beta\ r$ could be defined directly for x, $*e$, and $e{\rightarrow}f$ without it). Thus, in essence, the cell evaluation judgment captures the additional complexity of internal pointers (from $\&e$) and nested structures (from $e.f$). In fact, we include the FIELDDEREF rule even though the expression $e{\rightarrow}f$ is a synonym for $(*e).f$ as in C to emphasize this point. Now, the transfer function for an update can be captured extremely concisely with the following forward Hoare rule:

$$\frac{e \downarrow (\alpha\ \ell \mapsto \beta\ r) \qquad e' \Downarrow \beta'\ r'}{\{\alpha\ \ell \mapsto \beta\ r\}\ \ e := e'\ \ \{\alpha\ \ell \mapsto \beta'\ r'\}}$$

Note that this one rule captures updates to variables and fields given by arbitrary access paths involving '$*$', '.', and '&' (but ignoring size constraints).

Analyzing C-Style Dynamic Memory Management. Intuitively, the transfer function for **free** should simply delete the outgoing points-to edges from the pointer being freed. However, according to C standard, **free** can only be called on pointers to the base address of an allocated block previously returned by **malloc**. For instance, in the code below, the pointer value `&y` cannot be passed to **free** because it was not returned by **malloc**:

```
S* x = (S*)malloc(sizeof(S)); S y = *x;
free(x) /* ok */; free(&y) /* fails */;
```

The address-of operator $\&e$ permits the creation of pointer values that are not necessarily returned by **malloc**. The analysis must therefore track the nodes that represent the base address of an allocated block along with those edges that make up the block. Such a "tag" for allocated blocks is an example of a property that naturally applies to nodes.

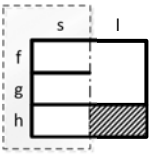
3.2 Compiler Dependencies Induced by Union Types

Thus far, we have focused on compiler independent modeling. However, one prevalent use of C is to access low-level features that are necessarily dependent on the compiler implementation. For example, a program may rely on sizes (e.g., `int` being 32-bits), address arithmetic, or a particular struct/object layout. In this section, we describe language features that are often used in a way dependent on the compiler implementation. Others have also realized that sometimes it is necessary to analyze code in a compiler-dependent manner (e.g., [20]).

Untagged Unions and Overlapping Cells. One such instance is dependence on multiple access paths mapping to the same memory location, which may occur

```
1 union {
2   struct { int f; int g; int h; } s;
3   long long l;
4 } x, y;
5 x.s.f = ...;
6 x.s.h = ...;
7 y.l = x.l;
```

(a) Union-manipulating code.



(b) An instance of the union type.

Fig. 7. An example to illustrate compiler dependence with C-style unions

with untagged unions in C. For example, consider the C code in Fig. 7. There are several questions that are compiler dependent. Will the write to `x.s.h` on line 6 modify `x.l`? Will the write on line 7 copy `x.s.f` and `x.s.g` to `y.s.f` and `y.s.g`? Should the read from `x.l` be allowed? Strictly according to the C specification, we might say no [16], but we might also want to analyze programs that use such assumptions. Furthermore, regardless of layout, a suitable representation must allow us to determine that the write to line 5 modifies `x.l` but not `x.s.g` and `x.s.h`. Note that the same kind of issues arise with type conversions and pointer arithmetic.

For such reasoning, we must expose byte-level offsets and field sizes. Offset expressions ℓ are now access paths p_{ath} as before (for compiler-independent accesses) or a path followed by a pair of a byte-level numeric offset o and an access path (for compiler-dependent ones) as defined in the inset. The byte-level numeric offset is given by the compiler to correspond to the bundled access path. To expose field sizes, we annotate points-to edges with the size of the memory cell it represents as a compiler-provided integer sz (where necessary). We thus have edges of the following form: $\alpha\ \ell \mapsto \beta\ r$ or $\alpha\ \ell \overset{sz}{\mapsto} \beta\ r$. Note that this size information is a property of the memory cell and thus appears on the edge and not on the nodes.

$$\begin{aligned} \ell &::= p_{ath} \mid p_{ath} + o/p_{ath} \\ r &::= \ell \\ p_{ath} &::= \varepsilon \mid p_{ath}@f \end{aligned}$$

These offset expressions allow us to express untagged unions directly. All fields of a union will have the same numeric offset but different access paths. Conceptually, with additional compiler-specific information about sizes, the analyzer can ensure that writes to any field will overwrite and remove the information about overlapping fields. Reading from a field that was not previously written can also be detected and either throw an error or rely on compiler dependent behaviors to interpret the data from the other field depending on the desired model. However, having a points-to edge for each union field would violate our representation invariant that all edges in the graph are separately conjoined. Union fields share the same concrete memory cells and thus are clearly not separate. What we want is some amount of local sharing (or use of non-separating conjunction) but we want to keep this additional complexity isolated.

To address this issue, we first introduce *memory region values* that correspond to memory regions on which there exists multiple views. In essence, to represent a union, the graph contains a points-to edge for the entire union and whose

contents are points-to edges for the substructure. We notate a memory region value as follows: $[\ +o_1/p_{ath_1} \overset{sz_1}{\mapsto} \gamma_1\ r_1 \mid \cdots \mid +o_n/p_{ath_n} \overset{sz_n}{\mapsto} \gamma_n\ r_n\ ]$.

Example 4 (Representing a Union). With memory region values, we represent an instance of the union type described in Fig. 7b (e.g., x) as follows:

$$\alpha \overset{12}{\mapsto} \beta \ \ \wedge \ \ \beta = [\ +0/@\mathsf{s}@\mathsf{f} \overset{4}{\mapsto} \gamma \mid +4/@\mathsf{s}@\mathsf{g} \overset{4}{\mapsto} \delta \mid +8/@\mathsf{s}@\mathsf{h} \overset{4}{\mapsto} \varepsilon \mid +0/@\mathsf{l} \overset{8}{\mapsto} \eta\ ]$$

Note the similarity of memory region values with record values for nested structures (Sect. 3.1). In particular, we can interpret memory region values as follows in separation logic:

$$\begin{aligned}&\alpha\ \ell \overset{sz}{\mapsto} \beta \ \ \wedge \ \ \beta = [\ +o_1/p_{ath_1} \overset{sz_1}{\mapsto} \beta_1\ r_1 \mid \cdots \mid +o_n/p_{ath_n} \overset{sz_n}{\mapsto} \beta_n\ r_n\ ]\\ &\quad \overset{\text{def}}{=} (\alpha\ \ell\ +o_1/p_{ath_1} \overset{sz_1}{\mapsto} \beta_1\ r_1 * \mathsf{true}) \ \wedge \ \cdots \ \wedge \ (\alpha\ \ell\ +o_n/p_{ath_n} \overset{sz_n}{\mapsto} \beta_n\ r_n * \mathsf{true})\end{aligned}$$

We write $\ell\ +o'/p_{ath}'$ for the concatenation of paths and offsets as appropriate (i.e., instantiating ℓ, we have two cases: $p_{ath}\ +o'/p_{ath}' \overset{\text{def}}{=} p_{ath}\ +o'/p_{ath}'$ and $+o/p_{ath}\ +o'/p_{ath}' \overset{\text{def}}{=} +(o+o')/p_{ath}\ p_{ath}'$ where $p_{ath}\ p_{ath}'$ is the concatenation of paths). Observe that the memory region edge (whose contents is a memory region value) encloses the complex sharing and can coexist with edges that do not have numeric offsets or even sizes. Also, note that with numeric offsets, it is tempting to compile away access paths into numeric offsets. However, doing so throws away useful object-level information.

At the same time, the above definition is not completely satisfactory from the point of view of representing unions entirely in the separating shape graph because of the use of non-separating conjunction ($\wedge$). Specifically, we desire a rule to push union edges of a memory region value into the graph (like for record values in Example 3). We observe that this use of non-separating conjunction is local to a node (i.e., it involves only outgoing points-to edges from α in the above). Thus, we simply need a mechanism to mark that a set of points-to edges from a node may share the same concrete memory cells (e.g., the edges on the right-side of the definition would be marked as such a set). The analyzer must then consider all of the points-to edges in the set simultaneously whenever one of them is updated. For example, the separating shape graph at program point 8 in Fig. 4 shows the edge labeled node@uni@s that is in fact one such shared edge. In the figure, we have elided other edges in its shared set; being more explicit, there are four edges in the set with the following addresses: β @node +0/@uni@s, β @node +0/@cst@value, β @node +0/@bin@l, and β @node +4/@bin@r. Note that one elegant way to keep track of which edges may share the same memory region is to apply the idea of fractional permissions [2,3]. Intuitively, reading from a union field requires only shared permission ($0 < \textit{permission} < 1$) but writing to a union field requires exclusive permission ($\textit{permission} = 1$) to ensure all other union fields are updated appropriately.

Compiler Independent Uses of Unions. Not all uses of unions depend on the compiler implementation like in Example 4. We may instead conservatively model

that all branches of a union may overlap (e.g., `x.s.h` and `x.l` is not known to be disjoint). To do so, observe that the size and offsets need not correspond to sizes and offsets in bytes as long as they conservatively model overlap. For example, consider the following that conservatively approximates Example 4:

$$\alpha \overset{1}{\mapsto} \beta \quad \wedge \quad \beta = [\ +0/@\mathtt{s}@\mathtt{f} \overset{0.3}{\mapsto} \gamma \mid +0.3/@\mathtt{s}@\mathtt{g} \overset{0.3}{\mapsto} \delta \mid +0.6/@\mathtt{s}@\mathtt{h} \overset{0.4}{\mapsto} \varepsilon \mid +0/@\mathtt{l} \overset{1}{\mapsto} \eta\ ]$$

Here, we say $\alpha \mapsto \beta$ is of "unit size" and where the union fields `s` and `l` occupy the entire region; then, the structure fields within `s` divide up the region.

4 A C Memory Model as Separating Shape Graphs

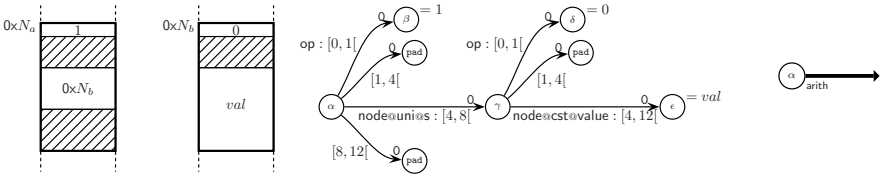
In this section, we formalize a static analysis abstraction, instantiating the shape graphs introduced in Sect. 3, with explicit byte-level offsets and sizes.

A classical definition for memory states **Mem** is a finite map from values into values, that is, to let $\mathbf{Mem} = \mathbf{Val} \rightharpoonup_{\text{fin}} \mathbf{Val}$ where **Val** denotes machine values. However, this definition does not directly capture the properties we want to express and abstract. First, we need a detailed description of memory with fields, addresses, and sizes. Second, we need to account for memory management—we need to know for each byte, in which block it was allocated. Therefore, we adopt a lower-level and more precise definition here. Our definition is based on a notion of contiguous regions, that is, unbroken chunks of memory. A memory state is specified as a set of allocated regions, a subdivision of these chunks into fields, and a value mapping for each element of this subdivision. A contiguous region r is defined by its base address ba and its size in bytes sz. We use subscripts to indicate the region of a particular component (e.g., ba_r for the base address of region r). A region r then covers the range of addresses $R = [ba_r, ba_r + sz_r - 1]$. We say that regions r and r' are disjoint if and only if their ranges are disjoint (i.e., $R_r \cap R_{r'} = \emptyset$). A concrete memory state σ is a tuple (m, s, c) composed of the following:

- A *table of allocated memory chunks* m, which we model with a set of regions. These chunks represent allocation with **malloc** or on the stack.
- A *subdivision* s, which is a set of regions such that for all $r, r' \in s$, regions r and r' are disjoint, and for all $r \in s$, there exists an allocated memory chunk $k \in m$ such that $R_r \subseteq R_k$.
- A *content* function c, which consists of a function from s into content values such that for all regions $r = (ba_r, sz_r) \in s$ where $c(r)$ is defined, it denotes a value of sz_r bytes.

Two concrete memory states are equivalent if and only if their content functions describe the same address to byte mapping. In the following, we reason up to this equivalence and consider two equivalent memory states equal.

Figure 8a depicts an excerpt from a concrete store, which contains an `Arith` structure from Fig. 4 that represents the expression $-val$. Note that the `Arith` expression nodes do not have the same layout due to the union field in use. Each



(a) Concrete store. (b) Unfolded graph. (c) Folded graph.

Fig. 8. Representation of an `Arith` object from Fig. 4

bold block corresponds to an allocated region (of size 12) and is partitioned into smaller regions that correspond to each field.

We can now describe the concretization of a separating shape graph $M \in \mathbb{M}^\sharp$ as a set of concrete stores σ. Recall from Fig. 5 that the analysis state contains an element $P \in \mathbb{P}^\sharp$ that tracks properties of values. For example, we may need numeric information (e.g., $\alpha = 42$) or to tag a value with its size (e.g., $\text{sizeof}(\beta) = 8$). To express the denotation of such an abstract element, we need to take into account the mapping of symbolic values $\alpha \in \mathbf{Val}^\sharp$ into values (e.g., addresses, integers, record values). This mapping is given by *valuations* $\nu : \mathbf{Val}^\sharp \to \mathbf{Val}$, which describes a physical mapping. Formally, the concretization is defined by

$$
\begin{array}{lcll}
\boldsymbol{\gamma}_{\mathbb{M}} & : & \mathbb{M}^\sharp & \to \quad \mathcal{P}(\mathbf{Mem} \times (\mathbf{Val}^\sharp \to \mathbf{Val})) \\
\boldsymbol{\gamma}_{\mathbb{P}} & : & \mathbb{P}^\sharp & \to \quad \mathcal{P}(\mathbf{Val}^\sharp \to \mathbf{Val}) \\
\boldsymbol{\gamma} & : & \mathbb{D}^\sharp & \to \quad \mathcal{P}(\mathbf{Mem}) \\
 & \stackrel{\text{def}}{=} & (M, P) & \mapsto \quad \{\, \sigma \in \mathbf{Mem} \mid \exists \nu.\, \nu \in \boldsymbol{\gamma}_{\mathbb{P}}(P) \;\wedge\; (\sigma, \nu) \in \boldsymbol{\gamma}_{\mathbb{M}}(M) \,\}
\end{array}
$$

where $\boldsymbol{\gamma}_{\mathbb{M}}$ and $\boldsymbol{\gamma}_{\mathbb{P}}$ are the concretization functions for separating graphs and elements of the base domain, respectively. The function $\boldsymbol{\gamma}$ is the concretization for the product domain $\mathbb{D}^\sharp$ of separating shape graphs and the base domain; note that the valuation ν connects the concretizations of the components M and P. In the following, we detail the main features of $\boldsymbol{\gamma}_{\mathbb{M}}$, including how we concretize edges, disjoint regions, contiguous region summaries, and non-contiguous region summaries. We discuss these aspects in the context of the shape graphs in Fig. 8b and Fig. 8c that abstract the concrete store shown in Fig. 8a.

Concretizing Points-To Edges. A points-to edge models one memory cell with an address and contents. The address of the memory cell is represented by a base address—it's source node—and optionally an offset expression ℓ. Similarly, the contents of a memory is given by a base address—it's target node—and an optional offset. In the following, we use only byte-level offsets (i.e., $+o$), as symbolic access paths simply concretize to byte-level offsets in a compiler-dependent manner (e.g., $@f$ lowers to $+\,\text{offset}(f)$). Similarly, we assume points-to edges have been annotated with the size of the cell they represent (e.g., by looking at field types in a compiler-dependent way).

Definition 1 (Concretization of Points-To Edges). *A* points-to edge *is given by a* source $(\alpha, o) \in \mathbf{Val}^\sharp \times \mathbb{N}$*, a* destination $(\alpha', o') \in \mathbf{Val}^\sharp \times \mathbb{N}$*, and a* size $sz \in \mathbb{N}$*. We notate such an edge as* $\alpha{+}o \overset{sz}{\mapsto} \alpha'{+}o'$*. The concretization on edges for points-to* $\boldsymbol{\gamma}_{\mathcal{E}}(\alpha{+}o \overset{sz}{\mapsto} \alpha'{+}o')$ *is defined as follows:*

$$(\sigma, \nu) \in \boldsymbol{\gamma}_{\mathcal{E}}(\alpha{+}o \overset{sz}{\mapsto} \alpha'{+}o') \quad \textit{if and only if} \quad \begin{array}{l} \sigma = (m, s, c) \quad \textit{and} \\ s = \{\, (\nu(\alpha) + o, sz) \,\} \quad \textit{and} \\ c(\nu(\alpha) + o, sz) = \nu(\alpha') + o' \end{array}$$

That is, the concrete memory is a single region with base address $\nu(\alpha) + o$ *and size* sz *with contents* $\nu(\alpha') + o'$*. Note that* ν *should interpret* α' *as an* sz*-bytes value (while we keep this implicit here, type and size information of values should be tracked in practice).*

Offsets o, o' are integers. Hence, this presentation allows for a straightforward handling of field-level pointer arithmetic. For instance in Fig. 8b, the edge drawn from α to γ is annotated with the range $[4, 8[$ and the target offset `0`: it corresponds to a memory cell of size 4, with base address $\nu(\alpha) + 4$, and with contents $\nu(\gamma)$ and thus concretizes to part of the concrete store shown in Fig. 8a assuming $\nu(\alpha) = \mathtt{0x}N_a$ and $\nu(\gamma) = \mathtt{0x}N_b$.

Concretizing Disjoint Memory Chunks. Recall from Sect. 2 and Sect. 3 that in a separating shape graph, distinct edges stand for disjoint chunks of memory (assuming unions are represented as memory region values). Thus, the concretization of a graph is the union of the concretizations of each edge with a disjointness or separation constraint. This constraint is analogous to formulas in separation logic conjoined with $*$; however, to treat allocated memory explicitly, we give direct a formalization here. For the moment, assume that the graph M is fully unfolded (i.e., contains only points-to edges), then we define $\boldsymbol{\gamma}_{\mathbb{M}}(M)$ as the set of all $(\sigma, \nu) \in \mathbf{Mem} \times (\mathbf{Val}^\sharp \to \mathbf{Val})$ where $\sigma = (m, s, c)$ and such that

- For each node α in graph M, if α is the base address of a memory region of size sz, then region $(\nu(\alpha), sz)$ belongs to m. In other words, the concrete memory state σ has an allocated region at $\nu(\alpha)$ of size sz.
- There exists a family of memory states $(\sigma_e)_{e \in M}$ such that

$$\sigma = \text{*}\{\sigma_e \mid e \in M\} \;\text{ and }\; \sigma_e \in \boldsymbol{\gamma}_{\mathcal{E}}(e) \text{ for all } e \in M$$

  where we write $e \in M$ for an edge e in graph M and overload $\sigma_1 * \sigma_2$ on concrete states to mean the combining of disjoint memories σ_1 and σ_2. That is, memory state σ can be partitioned into a set of memory states that are the contributions of each of the edges of the graph.

A C **struct** consists of a set of contiguous cells. For instance, the concrete store presented in Fig. 8a is abstracted by the unfolded graph of Fig. 8b where each edge corresponds to a subregion of the concrete store. Note that to be completely explicit, some edges correspond to padding generated as part of the compiler-dependent lowering (no information is ever available about the content nodes).

In the remainder of this section, we consider the concretization of memory region values introduced in Sect. 3.2 to capture C-style unions (Sect. 4.1) and of inductive summaries (Sect. 4.2). Section 4.3 sums up the analysis operations using this instantiation of separating shape graphs.

4.1 Concretizing Contiguous Regions

Arrays and Points-To Edges over Non-Constant Ranges. Arrays correspond to contiguous sequences of bytes in memory, so sized points-to edges can be used to capture array values. For a fixed-length array, we can split it into a points-to edge for each cell, but we can also choose to represent it as one large cell and abstract its contents with a dedicated array domain (e.g., [12,13,15]) as part of $\mathbb{P}^\sharp$. With a slight extension to allow field sizes and offsets to be expressed symbolically (i.e., in terms of symbolic values), we can also model non-fixed-length arrays as one large region or some finite number of chunks. The base domain $\mathbb{P}^\sharp$ should express range and congruence constraints about that offset, like in Miné [20]. This representation is similar in purpose to iterated separating conjunction [22], but we generally want the entire contents to be modeled as a single value.

In certain cases, using one large cell may be desirable, as existing array abstractions can be re-used together with our shape abstraction. Thus, we can avoid a need to reason precisely about indexing expressions in the shape domain. At the same time, this choice potentially limits the interaction between the domains making it more difficult to analyze code that, for example, have an inductive structure using arrays of pointers.

Untagged Unions and Overlapping Regions. As alluded to Sect. 3.2, memory region values are key to capturing untagged unions or in general multiple regions for the same memory region.

Definition 2 (Concretization of Memory Region Values). *A* multi-view points-to edge *is a points-to to a memory region value, that is,* $\alpha{+}o \overset{sz}{\mapsto} \beta$ *and* $\beta = [\ o_1 \overset{sz_1}{\mapsto} \alpha'_1\ o'_1 \mid \cdots \mid o_n \overset{sz_n}{\mapsto} \alpha'_n\ o'_n\ ]$ *such that for all* i *such that* $1 \leq i \leq n$, $o_i{+}sz_i \leq sz$. *The concretization of such a family of edges is the set of pairs* (σ, ν) *such that* $\sigma = (m, s, c)$ *where* $s = \{\ (\nu(\alpha){+}o, sz)\ \}$ *and* $\mathbf{read}(c, \nu(\alpha){+}o{+}o_i, sz_i) = \nu(\alpha'_i) + o'_i$ *(for all* i*). The operation* $\mathbf{read}(c, v, sz)$ *stands for the* sz*-bytes value that can be read in contents* c *from address* v. *In other words, the concrete memory is a single region given by the points-to edge* $\alpha{+}o \overset{sz}{\mapsto} \beta$ *but whose contents are also described by each of the views of* β.

4.2 Summarizing Complex Regions Using Inductive Definitions

Recall that we summarize non-contiguous regions of unbounded size with *checker edges* that correspond to inductive structures. As in our prior work [6,5], we take advantage of user-supplied inductively-defined checkers c and generate summaries that correspond to complete and partial structures. In particular, a *checker edge* $(\alpha{+}o).c(\delta)$ is an instance of an inductive checker definition c, and

a *segment edge* $(\alpha{+}o).c(\delta) \ast\!\!= (\alpha'{+}o').c'(\delta')$ is a partial derivation of checker c from $\alpha{+}o$ up to $\alpha'{+}o'$ and expecting checker c'.

We give here an *indirect* definition of the concretization of graphs containing summary edges: in a first step, we unfold shape graphs into fully unfolded shape graphs with no summary edges; in a second step, we concretize these using the previously defined concretization. This definition captures the same notion of inductively-defined regions as our previous definition [6,5], yet we take this indirect approach here since it extends more cleanly to the case where we take allocated regions into account. As a notation, we write $(M, P) \rightsquigarrow (M', P')$ to mean that the pair of graph and base domain element (M', P') can be obtained from (M, P) by unfolding one summary edge in M once. As inductive checkers include data constraints, unfolding updates both the graph and the base domain element.

Definition 3 (Concretization of a Graph with Summary Edges). *The concretization* $\boldsymbol{\gamma}_{\mathbb{M}}(M)$ *is the set of pairs* (σ, ν) *such that* $(M, \top) \rightsquigarrow^* (M', P')$*,* M' *is fully unfolded,* $\sigma \in \boldsymbol{\gamma}_{\mathbb{M}}(M')$*, and* $\nu \in \boldsymbol{\gamma}_{\mathbb{P}}(P')$*. Note that we write* $\top$ *for the top element of the base domain (i.e., no data constraints) and* $\rightsquigarrow^*$ *for the reflexive-transitive closure of* $\rightsquigarrow$*.*

Returning to the syntax tree example from Fig. 4, a user-supplied checker for arith may specify that op serves as the discriminator:

```
t.arith() :=
  if (t.op = 0) then true
  else if (t.op = 1) then t.node.uni.s.arith()
  else if (t.op >= 2) then t.node.bin.l.arith() and t.node.bin.r.arith()
```

(i.e., 0 is for constants, 1 is a unary operator, and ≥ 2 are binary operators). This checker translates to the following low-level definition with compiler-specific offsets and sizes made explicit (which could be obtained from the C types):

$$
\begin{aligned}
&\pi.\mathsf{arith}() := \\
&\quad \langle\, \pi@\mathsf{op} \overset{1}{\mapsto} \beta \ast \pi@\mathsf{node} \overset{8}{\mapsto} \gamma, \mathbf{alloc}(\pi, 12) \wedge \beta = 0 \wedge \\
&\qquad \gamma = [+0/@\mathsf{cst}@\mathsf{value} \overset{8}{\mapsto} \delta_1 \mid +0/@\mathsf{uni}@\mathsf{s} \overset{4}{\mapsto} \delta_2 \mid +0/@\mathsf{bin}@\mathsf{l} \overset{4}{\mapsto} \delta_3 \mid +4/@\mathsf{bin}@\mathsf{r} \overset{4}{\mapsto} \delta_4] \,\rangle \\
&\quad \vee \langle\, \pi@\mathsf{op} \overset{1}{\mapsto} \beta \ast \pi@\mathsf{node} \overset{8}{\mapsto} \gamma \ast \delta_2.\mathsf{arith}(), \mathbf{alloc}(\pi, 12) \wedge \beta = 1 \wedge \gamma = \ldots \,\rangle \\
&\quad \vee \langle\, \pi@\mathsf{op} \overset{1}{\mapsto} \beta \ast \pi@\mathsf{node} \overset{8}{\mapsto} \gamma \ast \delta_3.\mathsf{arith}() \ast \delta_4.\mathsf{arith}(), \mathbf{alloc}(\pi, 12) \wedge \beta \geq 2 \wedge \gamma = \ldots \,\rangle
\end{aligned}
$$

The predicate **alloc** expresses that a base address is an allocated region of a particular size. We note that Fig. 8b is one of the unfolded versions of Fig. 8c; that is, Fig. 8c abstracts the concrete store of Fig. 8a.

4.3 Shape Analysis for Compiler-Dependent C

Given a concrete operation Φ : **Mem** $\rightarrow$ **Mem**, the corresponding abstract transfer function $\Phi^\sharp : \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$ should be sound, that is, for all $D \in \mathbb{D}^\sharp$ and for all $\sigma \in \boldsymbol{\gamma}(D)$, it is the case that $\Phi(\sigma) \in \boldsymbol{\gamma}(\Phi^\sharp(D))$. In other words, performing the operation at the abstract level does not lose any concrete behavior.

Transfer Functions and Materialization. To reflect assignment statements and conditional guards, transfer functions evaluate expressions to cells and values using the rules given in Fig. 6 to determine which edges should be modified. However, in many cases, the edges corresponding to subexpressions are not readily available in shape graphs. They need to be materialized, that is, we need to concretize part of the summarized regions so that the appropriate points-to edges are revealed. Since materialization is a partial concretization operation, we now have two ways to materialize: for non-contiguous regions (Sect. 4.2) and for contiguous regions (Sect. 4.1). The former case corresponds to unfolding an inductive summary and is described in detail in our previous work [5]; the latter case corresponds to splitting a subpart of a points-to edge and is new to our framework. To describe this new materialization operation, we write $\mathbf{extract}_{[i,j[}(\alpha)$ for the operation that extracts bytes i to j from the value represented by α. Now, if $0 \leq sz_0 < sz$, then the following pair of edges and constraints can be materialized from the edge $\alpha{+}o \overset{sz}{\mapsto} \alpha'$:

$$\alpha{+}o \overset{sz_0}{\mapsto} \alpha'_0 \;*\; \alpha{+}(o+s_0) \overset{sz-sz_0}{\mapsto} \alpha'_1$$
$$\text{where} \quad \alpha'_0 = \mathbf{extract}_{[0,s_0[}(\alpha') \quad \text{and} \quad \alpha'_1 = \mathbf{extract}_{[s_0,s[}(\alpha')$$

The two last constraints are represented (in a conservative way) in $\mathbb{P}^\sharp$. This rule allows, for example, to materialize a single array cell from a whole array value.

Memory Management Operators. To model a successful call to **malloc**, the analysis creates a fresh memory region value β tagged with the size of the allocated area; it then creates a points-to edge of that size to β. To analyze a call to **free**, we need to materialize the entire region to free based on the allocated-size predicate on the node. We then check that the region to free was indeed allocated before discarding the edges corresponding to the region. The pointer to the address of the freed block becomes dangling (i.e., all outgoing edges are removed). Parkinson [21] has also described this need to track allocated regions.

Widening. To enforce termination, we use a *widening* operator, which was extensively described in our prior work [6,5]. What is particularly interesting is that this operator requires minimal changes to accommodate the new kinds of edges introduced in this paper. Intuitively, the widening relies only on the graph structure, which is conserved by our extensions in this paper. It is sound (i.e., computes an over-approximation of concrete joins) and terminating (there is no infinite, non-converging sequence of widening iterates).

5 Implementation and Timing Results

We have extended the memory model of Xisa to reflect the features introduced in Sect. 4, including support for nested structures, pointers to internal fields, numerical offsets and sizes, memory region values, and base address of allocated blocks (to check **malloc** and **free** in a sound manner). The overall structure of

Table 1. Benchmark results for verifying shape preservation. We show the size of the benchmark in lines of code (number of lines of the relevant function), the analysis time, the maximum number of disjuncts at any program point (Peak Disj.), and the maximum number of iterations at any point (Iter.).

Benchmark	Size (loc)	Time (sec)	Peak Disj. (num)	Iter. (num)
traversal	20	0.036	8	2
eval	70	0.060	24	2x2
remsub	37	0.116	8	2
distribute	41	0.144	14	2
move_neg_up	120	0.488	38	2

unfolding and folding (widening) algorithms remained largely unchanged; there were only small, local extensions to deal with the new annotations on points-to edges. Code that was analyzable by the previous Xisa implementation is analyzable with this finer model. Support for arrays does not yet exist, primarily because it would require a more expressive numerical domain $\mathbb{P}^\sharp$ and extensions to the base domain interface.

Table 1 shows some implementation results that require this refined memory model. These examples are algorithms that traverse and/or modify in place a syntax tree structure like the one shown in Fig. 4. They evaluate or simplify arithmetic expressions (e.g., by distributivity) and delete or create new nodes as needed. In the table, we show analysis times, the maximum number of disjuncts at any program point, and the number of widening iterations needed in each loop (in the case of the "eval" example, we give numbers for each nested loops). The low values for number of iterations provides evidence that our widening operator enforces quick convergence while retaining precise results. We note that the peak number of disjuncts is rather high in the last example. This high number is due to the presence of nested if-statements that lead to successive unfolding of several levels of checker edges. Since we only try to collapse disjuncts at widening points, this implementation choice results in an exponential number of disjuncts in short code sections. Better heuristics to control the maximal number of disjuncts could improve performance, though we leave this to future work.

6 Related Work

The use of shape graphs for approximating unbounded structures dates back to at least Jones and Muchnick [17]. Their design and use have formed the basis of several steps in the development of shape analysis. Sagiv *et al.* [23] defined an early version of materialization with shape graphs that was subsequently refined in TVLA [24] with the perspective of "partial concretization" and the ability to simultaneously express both *may* and *must* relations between objects. A line of subsequent work has looked at compacting this representation (e.g., by merging similar graphs [19]). Traditionally, shape graphs have been applied

on Java-like structures using the the "object-as-node" paradigm. Very recently, Kreiker *et al.* [18] have formulated an extended memory model in the TVLA framework to reason about pointers to nested structures. They describe shape graph models that capture nested structures and internal pointers using both "field-as-node" and "object-as-node" paradigms. In contrast, separating shape graphs take a "cell-as-edge" approach inspired by separation logic [22], which we use to separate object or value-level properties on nodes from field or component-level properties on edges.

There has also been a line of work that builds shape analyzers around formulas in separation logic (e.g., [11,14,6]). In the last few years, significant progress has been made in handling realistic C code. For example, Berdine *et al.* [1] handle composite data structures, such as lists of lists, and Yang *et al.* [26] have looked at a ≈10,000 line device driver. Nonetheless, the focus has been on Java-like structures (i.e., limited reasoning on internal pointers or layout dependent features). One exception is Calcagno *et al.* [4] that have described a low-level analyzer with pointer arithmetic inside memory blocks.

There are also program analyzers, such as Miné [20], that address many low-level aspects of C, including unions and pointer casts, but they are not typically concerned with dynamic memory allocation and unbounded structures as in shape analysis. Another class of tools focuses on being as concrete as possible potentially trading off some automation or exhaustiveness. We take a different angle where we want a representation that supports user-guided abstraction. The HAVOC tool [7] combines reachability predicates with pointer arithmetic reasoning and has been applied to verify low-level properties of system drivers [9]. Clarke *et al.* [8] give a low-level encoding of C features for model checking. Xie and Aiken [25] perform exact bit-level encoding with bounded symbolic execution.

7 Conclusion

In this paper, we propose separating shape graphs as an abstraction that can handle typical, high-level data types and low-level aspects of C in a compositional manner. From the analysis point of view, the main result is that existing algorithms for unfolding and widening of shape abstractions are mostly unaffected in this extended framework.

Acknowledgments. We thank Jörg Kreiker, Antoine Miné, Hongseok Yang, Matthew Parkinson and Peter O'Hearn for stimulating discussions.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Principles of Program. Lang., POPL (2005)
3. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694. Springer, Heidelberg (2003)

4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: Static Analysis, SAS (2006)
5. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: Principles of Program. Lang., POPL (2008)
6. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
7. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamaric, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)
8. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Program. Lang., POPL (2009)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Program. Lang., POPL (1977)
11. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
12. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Principles of Program. Lang., POPL (2005)
13. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Principles of Program. Lang., POPL (2008)
14. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: Program. Lang. Design and Implementation, PLDI (2007)
15. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Program. Lang. Design and Implementation, PLDI (2008)
16. Harbison III, S., Steele Jr., G.: A Reference Manual. Prentice Hall, Englewood Cliffs (2002)
17. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of LISP-like structures. In: Program Flow Analysis: Theory and Applications, vol. 4 (1981)
18. Kreiker, J., Seidl, H., Vojdani, V.: Shape analysis of low-level overlapping structures. In: Verif., Model Checking, and Abstract Interp, VMCAI (2010)
19. Manevich, R., Sagiv, M., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
20. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: Lang. Compilers Tools Embed. Syst., LCTES (2006)
21. Parkinson, M.: Local reasoning for Java. PhD thesis, U. of Cambridge (2005)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, LICS (2002)
23. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Trans. Program. Lang. Syst. 20(1) (1998)
24. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
25. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Principles of Program. Lang., POPL (2005)
26. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)