Automatic Inference of Access Permissions

Pietro Ferrara and Peter Müller

ETH Zurich, Switzerland
{pietro.ferrara,peter.mueller}@inf.ethz.ch

Abstract. Access permissions are used in several program verification approaches such as those based on separation logic or implicit dynamic frames to simplify framing and to provide a basis for reasoning about concurrent code. However, access permissions increase the annotation overhead because programmers need to specify for each program component which permissions it requires or provides. We present a new static analysis based on abstract interpretation to infer access permissions automatically. Our analysis computes a symbolic approximation of the permissions owned for each heap location at each program point and infers a constraint system over these symbolic permissions that reflects the permission requirements of each heap access in the program. The constraint system is solved using linear programming. Our analysis is parametric in the permission system and supports, for instance, fractional and counting permissions. Experimental results demonstrate that our analysis is fast and is able to infer almost all access permissions for our case studies.

1 Introduction

Verification techniques based on access permissions associate a permission with each heap location. A thread may access a location if and only if it has the access permission for that location. This rule enables the verification of concurrent programs since it guarantees the absence of data races (two threads cannot both have the access permission for a memory location) and allows one to reason about thread interleavings (if a thread has the permission for a location, no other thread can modify it). Permissions can be transferred between threads when a thread is forked or joined, or via synchronization primitives such as monitors. To support procedure-modular verification, permissions are often associated with procedure (or method) incarnations; permissions are then transferred not only between different threads but also between callers and callees of the same thread. In this case, permissions simplify framing because a method may modify at most those locations for which it has permission; all other locations are guaranteed to remain unchanged. Permissions are used for instance in separation logic [22] and implicit dynamic frames [25].

Fractional permissions [2] and counting permissions [1] refine the permission model by allowing a full permission to be split (repeatedly) into fractions or into any number of units, which can be re-composed into a full permission. Both fractional and counting permissions allow one to distinguish between read access

V. Kuncak and A. Rybalchenko (Eds.): VMCAI 2012, LNCS 7148, pp. 202–218, 2012.

```
class Coord {
  var x: int;
  var y: int;

  invariant acc(x) && acc(y)
```

```
method client() {
  acquire this;
  var oldX := x; var oldY := y;
  this.FlipH();
  assert x == −oldX;
  assert y == oldY;
  release this;
}
```

```
method FlipH()
requires acc(x)
ensures acc(x)
ensures x == −old(x)
{ x := −x; }
}
```

Fig. 1. An example illustrating access permissions

(requiring any non-zero permission) and write access (requiring full permission) and, thus, support parallel reads while still enforcing exclusive writes.

Fig. 1 illustrates the use of permissions in Chalice [14,15]. A full permission to a location e.f is denoted by acc(e.f), which corresponds to e.f $\mapsto$ _ in separation logic. Chalice associates permissions with method incarnations and with monitors. The precondition and postcondition of a method specify the permissions a method expects from its caller and provides to its caller, respectively. A monitor invariant specifies the permissions associated with a monitor. When a method acquires a monitor, these permissions are transferred from the monitor to the method, and returned to the monitor when it is released.

In Fig. 1, the monitor invariant of class Coord expresses that the monitor holds the permission to this.x and this.y when the monitor is not currently held by any thread. Method client obtains permissions to this.x and this.y by acquiring the monitor of this. Method FlipH requires permission to location this.x via its preconditions, and returns it via its postcondition. When client calls FlipH, the permission to this.x is passed to FlipH; the permission is returned when FlipH terminates. Method client can call FlipH passing and receiving back access permission to this.x. Both assertions in client verify. The first assertion is established by the call to FlipH; since the current thread holds the permission to x, no other thread could invalidate the property between the call and the assertion. The second assertion illustrates framing: since client does not pass its permission for y to FlipH, we can conclude that FlipH cannot modify y. Both assertions would not verify if they were placed after the release statement because then other threads could obtain permissions to x and y and invalidate the asserted properties.

A drawback of all permission systems is that they require programmers to annotate their programs with access permissions, which increases the annotation overhead significantly. To address this issue, we present a new static analysis based on abstract interpretation to infer access permissions automatically. In this paper, we focus on the inference of access annotations for pre- and postconditions as well as monitor invariants, but our analysis also supports loop invariants and abstract predicates with fold and unfold statements [20]. Our paper makes four technical contributions: (1) a representation of permissions with symbolic values, (2) an inference of constraints over these symbolic values, (3) an inference of annotations, which supports fractional and counting permissions, as well as the combination of both, and (4) an implementation and experimental evaluation of the analysis in Sample (Static Analyzer for Multiple Programming LanguagEs).

The experimental results show that our analysis is practical and effective. It infers permission annotations for all examples in the Chalice test suite in under three seconds, and the necessary annotations in all examples except for four that use recursive data structures, for which our heap abstraction is too coarse. We expect that more precise heap analyses like TVLA [23] will solve this issue.

Approach. Our approach is based on abstract interpretation [6], a theory for defining and soundly approximating the semantics of a program. The main components of our analysis are: (1) an *abstract domain* that is a complete lattice, (2) a *widening* operator to make the analysis convergent, and (3) an *abstract semantics* defined as a transfer function that, given a statement and an initial abstract state, defines the abstract state obtained after the statement.

We introduce a *symbolic value* for each location and each possible occurrence of an access permission in a pre- or postcondition, or monitor invariant. For instance, $Pre(\mathsf{C}, \mathsf{m}, \mathsf{x.f})$ represents the permission specified in the precondition of method m of class C for the location denoted by the path x.f. Using these symbolic values, the analysis infers a sound approximation of the access permissions that the current method incarnation has for any given heap location at any given program point. These *symbolic permissions* have the form $\sum a_i * \mathsf{v}_i + c$ where a_i is an integer number, c is a real, and v_i is a symbolic value. For instance, if method m's first statement acquires the monitor of this then its symbolic permission for a location x.f is $1 * Pre(\mathsf{C}, \mathsf{m}, \mathsf{x.f}) + 1 * MI(\mathsf{C}, \mathsf{x.f}) + 0$.

We then extract a set of *constraints* over the symbolic values, which reflect the permission rules of the verification technique. For instance, for an assignment to x.f, we introduce a constraint that the symbolic permission at this program point is equal to the full (write) permission. Our constraints are parametric in the permission system being used. We solve the constraints using linear programming and obtain a numerical access permission for each symbolic value. For simplicity, we assume here that the inference is run on un-annotated programs, but partial annotations could be easily represented as additional constraints.

Outline. Sec. 2 introduces the language supported by our analysis and the running example. Sec. 3 sketches our heap analysis. Sec. 4 defines the abstract domain and semantics to approximate access permissions. Sec. 5 explains how we infer permission annotations. Sec. 6 reports the experimental results. Sec. 7 discusses related work, and Sec. 8 concludes.

2 Language

We present our analysis for a class-based language with threads and monitors.

Programs. A program consists of a sequence of class declarations. Each class declares fields, a monitor invariant, and methods. A method declaration contains the method signature, pre- and postconditions, and a method body, which is given as a control flow graph of basic blocks. Each basic block consists of a sequence of statements. Different blocks are connected through edges that optionally contain a boolean condition to represent conditional jumps.

Table 1. Expressions and statements. We denote thread identifiers by t.

$$\mathsf{E} ::= \mathsf{x} \mid \mathsf{x.f}$$

$$\begin{array}{lllll} \mathsf{St} ::= \mathsf{x := E} & \mid \mathsf{x := new\ T} & \mid \mathsf{acquire\ x} & \mid \mathsf{t := fork\ x.m()} & \mid \mathsf{share\ x} \\ \quad \mid \mathsf{x.f := E} & \mid \mathsf{x.m()} & \mid \mathsf{release\ x} & \mid \mathsf{join\ t} & \end{array}$$

The expressions and statements of our language are summarized in Table 1. We omit uninteresting expressions such as boolean and arithmetic operators here. We adopt the share statement from Chalice; it associates a (non-reentrant) monitor with a previously thread-local object to make it available for locking. We omit Chalice's unshare statement since it does not affect permissions.

Specifications and Permissions. Specifications are expressed using the expressions of the programming language, an old-expression to let postconditions refer to prestate-values, and permission predicates. A *permission predicate* has the form $\mathbf{acc}(\mathsf{x.f,p})$, where p denotes a permission. With fractional permissions, p is a fraction between 0 and 1, with counting permissions, p is an integer (negative numbers are interpreted as a full permission plus p counting permissions), and in Chalice, p is an expression of the form $q\% + n \cdot \epsilon$, where q is a natural number between 0 and 100, n is an integer, and ϵ is a constant that denotes an infinitesimal permission, that is, an arbitrarily small, positive number. The percentage encodes fractional permissions, whereas the infinitesimal permission is used as a unit of counting permissions. Independently of the permission system, we abbreviate a full permission as $\mathbf{acc}(\mathsf{x.f})$. For sound verification, it is important that specifications are *self-framing*, that is, a specification may refer to $\mathsf{x.f}$ only if it has the permission to access $\mathsf{x.f}$. So $\mathbf{acc}(\mathsf{x.f})$ && $\mathsf{x.f} > 0$ is a valid specification, but $\mathsf{x.f} > 0$ is not. We enforce this requirement by generating constraints not only for heap accesses in code but also in specifications.

Permission Transfer. Permissions can be transferred between two method incarnations and between a method incarnation and a monitor. For modular verification, we can describe these transfers from the perspective of the method incarnation that is currently executing. We say that a method *exhales* a permission if it transfers the permission to another method or a monitor, that is, gives up the permission. It *inhales* a permission if the transfer happens in the other direction, that is, when the method obtains the permission. For instance, when a monitor is released, its invariant is exhaled, while the invariant is inhaled when the monitor is acquired. Exhaling a permission entails a proof obligation that the method actually has the permission to be transferred. We say that a method exhales or inhales a specification, if it exhales or inhales all permissions mentioned in the specification. The statements of our language transfer permissions as follows. Assignments involve no permission transfer. When creating an object, we inhale full permissions for all locations of the fresh object. When we call a method, we first exhale the precondition of the callee and then inhale its postcondition. When a method acquires a monitor, we inhale the monitor

```
class W1 {
  var c : Cell;
  method Inc()
  ensures c.c1==old(c.c1)+1
  {
    acquire c;
    c.x := c.x+1;
    c.c1 := c.c1+1;
    release c;
  }
}

class Cell {
  var x, c1, c2: int;
   invariant x==c1+c2;
}
```

```
class W2 {
  var c : Cell;
  method Inc()
  ensures c.c2==old(c.c2)+1
  {
    acquire c;
    c.x := c.x+1;
    c.c2 := c.c2+1;
    release c;
  }
}
```

```
class OwickiGries {
  method main() {
    var c := new Cell;
    share c;
    var w1 := new W1;
    w1.c := c;
    var w2 := new W2;
    w2.c = c;
    t1 := fork w1.Inc();
    t2 := fork w2.Inc();
    join t1;
    join t2;
    acquire c;
    assert c.x==2;
  }
}
```

Fig. 2. The Owicki-Gries example without permission annotations

invariant, and we exhale the invariant when the monitor is released and when we first share the object. When forking a thread, we exhale the precondition of the forked method, while we inhale the postcondition when joining.

Running Example. We illustrate our inference using Owicki and Gries's classical example [19], see Fig. 2. Two worker threads, implemented in classes W1 and W2 each increment a shared variable x (of class Cell) by 1. The client (method main) asserts that the effect of running both workers is to increment x by 2. The standard solution to proving this assertion requires ghost (that is, specification-only) variables; the ghost fields c1 and c2 store the contribution of each worker to the overall effect. These ghost variables are related to x in Cell's monitor invariant and also mentioned in the workers' postconditions. Therefore, to enforce self-framing specifications, the postconditions and the monitor invariant need some access permission to the ghost variables. This can be achieved by using fractional or counting permissions to split the permissions over the postconditions and the monitor invariant. The example then verifies: From the postconditions, we know that after the two join operations in main, c1 and c2 have each been increased by 1. From Cell's monitor invariant, which we assume after the acquire statement, we know that then x has been increased by 2; so since fields of new objects are initialized to zero-equivalent values, the assertion verifies. We will show how our analysis infers the permission annotation to enable this verification.

3 Heap Analysis

Since access permissions guard accesses to heap locations, our inference requires information about the heap, for instance, to decide whether two expressions may refer to the same heap location. The analysis that approximates properties of the heap is crucial for the effectiveness of the permission inference. However, since the heap analysis is not the main focus of this paper, we only sketch the main ideas and notations of the implemented heap analysis here, which is an extension of our earlier work [10]. We expect that more sophisticated heap analyses such as shape analysis [23] could be combined with our inference.

Our heap analysis abstracts objects in the concrete heap to *abstract nodes* (set $\overline{\mathsf{L}}$) in an abstract heap. The abstraction identifies each object with the program point where it is created. That is, it abstracts all concrete objects created at the same program point (for instance, inside a loop) by the same abstract *summary node*. The heap analysis works modularly, that is, analyzes each method separately. The initial heap for a method contains one abstract node for each method argument and abstract object reachable from an argument. If some of these objects may be aliases (that is, their types do not exclude that they refer to the same object), they are instead represented by one summary node. The function $\overline{\mathit{isSummary}} : \overline{\mathsf{L}} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ yields whether a node is a summary node, that is, whether it may represent more than one object.

Our permission inference uses type information for abstract nodes to determine which monitor invariant to inhale and exhale. The function $\mathit{class} : \overline{\mathsf{L}} \rightarrow \mathsf{C}$ yields the class of an abstract node (C is the set of class identifiers); for summary nodes, it returns the smallest superclass of the classes of each concrete object represented by the summary node.

The function $\mathit{fields} : \overline{\mathsf{L}} \rightarrow \wp(\mathsf{F})$ yields the set of fields of an abstract node, and the union of these sets for a summary node; F is the set of field identifiers. A path in Path is a sequence starting with a variable and followed by field identifiers. As usual, we denote by $\mathsf{x.f}$ the concatenation of variable x with field identifier f.

The heap analysis needs to define the semantics of expressions and statements in order to describe their effects on the abstract heap. The function $\overline{\mathbb{E}} : (\overline{\mathsf{H}}, \mathsf{E}) \rightarrow \overline{\mathsf{L}}$ evaluates an expression in an abstract heap and yields the resulting abstract node ($\overline{\mathsf{H}}$ is the set of abstract heaps). We assume a semantics of statements that tracks how each statement modifies the abstract heap. We do not present the heap semantics here, and we leave the heap modifications implicit in the abstract semantics for the permission inference.

4 Symbolic Permissions

In this section we present the symbolic values, which represent permission predicates in specifications, the abstract domain, the abstract semantics, and an unsound approximation that can improve the results of the inference.

4.1 Symbolic Values

Programs may contain permission predicates in pre- and postconditions and monitor invariants (we ignore loop invariants here, but our analysis supports them). We represent the permissions in these specifications by symbolic values (set $\overline{\mathsf{SV}}$). The access permission for a path $\mathsf{p.f}$ specified (1) by the precondition or postcondition of a method m in class C is represented by $\mathit{Pre}(\mathsf{C}, \mathsf{m}, \mathsf{p.f})$ or $\mathit{Post}(\mathsf{C}, \mathsf{m}, \mathsf{p.f})$, respectively, and (2) by the monitor invariant of class C is represented by $\mathit{MI}(\mathsf{C}, \mathsf{p.f})$. Since there could be many (possibly infinite) paths to a location, our semantics always considers a shortest path.

4.2 Abstract Domain

The *symbolic access permission* for a single abstract location, that is, field of an abstract node, could combine several symbolic values, since it could be the result of inhaling and exhaling different method specifications and monitor invariants. Therefore we represent the symbolic permission as the *summation* of symbolic values s_i multiplied by integer coefficients a_i (to represent how many times we have inhaled or exhaled a permission) and an integer constant c (to represent the full permission that is inhaled when an object is created). Formally, $\overline{\mathsf{AV}} = \{\sum_\mathsf{i} \mathsf{a}_\mathsf{i} * \mathsf{s}_i + \mathsf{c} \text{ where } \mathsf{a}_\mathsf{i}, \mathsf{c} \in \mathbb{R}, \mathsf{s}_i \in \overline{\mathsf{SV}}\}$.

On these summations we define a lattice structure. Fig. 3 formalizes the lattice operators. To be sound, we compute at each program point the access permissions the current method *surely* has, that is, it has in all possible executions. For this reason, the upper bound—which is used in the abstract semantics for joins in the control flow—of two symbolic permissions $\bar{\mathsf{l}}_1$ and $\bar{\mathsf{l}}_2$ is the *minimum* of $\bar{\mathsf{l}}_1$ and $\bar{\mathsf{l}}_2$. Since symbolic values represent non-negative values, a safe approximation of the minimum of two symbolic permissions is computed using the minimum of the corresponding coefficients a_i and of the integer constant c. Conversely, the lower bound is the *maximum* of $\bar{\mathsf{l}}_1$ and $\bar{\mathsf{l}}_2$, which is computed analogously. In the lattice order, symbolic permission $\bar{\mathsf{l}}_1$ is less or equal $\bar{\mathsf{l}}_2$ iff $\bar{\mathsf{l}}_1$ represents *greater* or equal permissions than $\bar{\mathsf{l}}_2$, that is, each coefficient and the constant in $\bar{\mathsf{l}}_1$ is greater or equal than the corresponding coefficient and the constant in $\bar{\mathsf{l}}_2$; this definition is in line with defining the upper bound as the minimum. We assume that each concrete permission system defines two constants, zero and full, to denote the zero-permission and full permission, respectively. The bottom element is any value greater than full or less than zero, that is, any invalid value for a permission. The top element is zero. Then the lattice can be defined by $\langle \overline{\mathsf{AV}}, \leq_{\overline{\mathsf{AV}}}, \mathsf{zero} - 1, \mathsf{zero}, \sqcup_{\overline{\mathsf{AV}}}, \sqcap_{\overline{\mathsf{AV}}} \rangle$. Note that this domain does not track disjunctive information like $\mathsf{b} \Rightarrow \mathsf{acc(x.f)}$, but it can be used inside other generic domains to obtain precise disjunctive information [18].

In the above domain, we have a finite number of symbolic values, but the integer coefficients could decrease indefinitely. Therefore, we need a widening operator to ensure the termination of the analysis. Our widening abstracts the symbolic access permission to zero if it is decreasing. This definition reflects that if a loop exhales permissions in each iteration, we approximate it assuming that no permission is left when the loop terminates because we do not know statically how many times the loop body will be executed. However, none of the examples we analyzed required widening because such loops are not common.

The abstract domain $\overline{\mathsf{PL}}$ tracks the symbolic access permissions at a given program point for each field of each abstract node. Therefore, its state is repre-

$$(\textstyle\sum_j a^1_j * \bar{\mathsf{s}}_j + c_1) \sqcup_{\overline{\mathsf{AV}}} (\sum_j a^2_j * \bar{\mathsf{s}}_j + c_2) = (\sum_j \min(a^1_j, a^2_j) * \bar{\mathsf{s}}_j + \min(c_1, c_2))$$
$$(\textstyle\sum_j a^1_j * \bar{\mathsf{s}}_j + c_1) \sqcap_{\overline{\mathsf{AV}}} (\sum_j a^2_j * \bar{\mathsf{s}}_j + c_2) = (\sum_j \max(a^1_j, a^2_j) * \bar{\mathsf{s}}_j + \max(c_1, c_2))$$
$$(\textstyle\sum_j a^1_j * \bar{\mathsf{s}}_j + c_1) \leq_{\overline{\mathsf{AV}}} (\sum_j a^2_j * \bar{\mathsf{s}}_j + c_2) = \mathsf{true} \Leftrightarrow c_1 \geq c_2 \wedge \forall j : a^1_j \geq a^2_j$$

Fig. 3. Lattice operators on $\overline{\mathsf{AV}}$

sented by a function that maps abstract locations to symbolic access permissions: $\overline{\mathsf{PL}} : (\overline{\mathsf{L}} \times \mathsf{F}) \rightarrow \overline{\mathsf{AV}}$. The lattice operators are defined as the functional extensions of the lattice operators of $\overline{\mathsf{AV}}$.

4.3 Abstract Semantics

The abstract semantics formalizes the effects of statements on symbolic permissions. It uses the helper functions in Fig. 4. $\overline{\mathit{reach}}(\bar{\mathsf{r}}, \overline{\mathsf{h}}, \overline{\mathsf{R}}, \mathsf{p})$ yields the set of abstract locations that can be reached from an abstract node $\bar{\mathsf{r}}$ in a heap $\overline{\mathsf{h}}$, without traversing the abstract nodes in $\overline{\mathsf{R}}$, and for each reachable abstract location a path through which it can be reached; this path is an extension of path p, through which node $\bar{\mathsf{r}}$ is reachable from some starting point. The set $\overline{\mathsf{R}}$ is used to discard alternative paths to the same abstract location. Note that the definition of $\overline{\mathit{reach}}$ is recursive. For each recursive application, we use function $\overline{\mathit{reach1}}$ to add all abstract locations that are reachable in one step, that is, by accessing a field of $\bar{\mathsf{r}}$. The recursion is well-founded since the heap domain contains a finite number of abstract nodes, and the set $\overline{\mathsf{R}}$ grows in each recursive application.

Function $\overline{\mathit{reach}}$ is used to extract all the abstract locations for which we *potentially* inhale or exhale permissions, together with a shortest path through which these permissions could be inhaled or exhaled. Function $\overline{\mathit{rep}}$ uses these abstract locations and paths to construct a symbolic value for each of them. Its last argument determines what kind of symbolic value we want to obtain.

$$
\begin{array}{l}
\overline{\mathit{reach}} : (\overline{\mathsf{L}} \times \overline{\mathsf{H}} \times \wp(\overline{\mathsf{L}}) \times \mathsf{Path}) \rightarrow \wp(\overline{\mathsf{L}} \times \mathsf{F} \times \mathsf{Path}) \\
\overline{\mathit{reach}}(\bar{\mathsf{r}}, \overline{\mathsf{h}}, \overline{\mathsf{R}}, \mathsf{p}) = \{(\bar{\mathsf{r}}_1, \mathsf{f}, \mathsf{p}) : (\bar{\mathsf{r}}_1, \mathsf{f}) \in \overline{\mathit{reach1}}(\overline{\mathsf{h}}, \mathsf{p}) \wedge \bar{\mathsf{r}}_1 \notin \overline{\mathsf{R}}\} \cup \\
\quad \cup \{(\bar{\mathsf{r}}_2, \mathsf{f}_1, \mathsf{p}_1) \in \overline{\mathit{reach}}(\bar{\mathsf{r}}_1, \overline{\mathsf{h}}, \overline{\mathsf{R}} \cup \downarrow_1 (\overline{\mathit{reach1}}(\overline{\mathsf{h}}, \mathsf{p})), \mathsf{p.f}) : (\bar{\mathsf{r}}_1, \mathsf{f}) \in \overline{\mathit{reach1}}(\overline{\mathsf{h}}, \mathsf{p})\}
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathit{reach1}} : (\overline{\mathsf{H}} \times \mathsf{Path}) \rightarrow \wp(\overline{\mathsf{L}} \times \mathsf{F}) \\
\overline{\mathit{reach1}}(\overline{\mathsf{h}}, \mathsf{p}) = \{(\bar{\mathsf{r}}, \mathsf{f}) : \overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{p}) = \bar{\mathsf{r}} \wedge \mathsf{f} \in \mathit{fields}(\bar{\mathsf{r}})\}
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathit{rep}} : (\wp(\overline{\mathsf{L}} \times \mathsf{F} \times \mathsf{Path}) \times \overline{\mathsf{SV}}) \rightarrow \wp(\overline{\mathsf{L}} \times \mathsf{F} \times \overline{\mathsf{SV}}) \\
\overline{\mathit{rep}}(\{(\bar{\mathsf{r}}_1, \mathsf{f}_1, \mathsf{p}_1), \cdots, (\bar{\mathsf{r}}_i, \mathsf{f}_i, \mathsf{p}_i)\}, \bar{\mathsf{s}}) = \{(\bar{\mathsf{r}}_1, \mathsf{f}_1, \bar{\mathsf{s}}_1), \cdots, (\bar{\mathsf{r}}_i, \mathsf{f}_i, \bar{\mathsf{s}}_i)\} : \forall \mathsf{j} \in [1..\mathsf{i}] : \\
\qquad\qquad\qquad\qquad \bar{\mathsf{s}}_j = \begin{cases} \mathit{MI}(\mathsf{c}, \mathsf{p_j.f_j}) & \text{if } \bar{\mathsf{s}} = \mathit{MI}(\mathsf{c}, \mathsf{p}) \\ \mathit{Pre}(\mathsf{c}, \mathsf{m}, \mathsf{p_j.f_j}) & \text{if } \bar{\mathsf{s}} = \mathit{Pre}(\mathsf{c}, \mathsf{m}, \mathsf{p}) \\ \mathit{Post}(\mathsf{c}, \mathsf{m}, \mathsf{p_j.f_j}) & \text{if } \bar{\mathsf{s}} = \mathit{Post}(\mathsf{c}, \mathsf{m}, \mathsf{p}) \end{cases} \}
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathit{inhS}} : (\overline{\mathsf{PL}} \times \overline{\mathsf{L}} \times \mathsf{F} \times \overline{\mathsf{SV}}) \rightarrow \overline{\mathsf{PL}} \\
\overline{\mathit{inhS}}(\overline{\sigma}, \bar{\mathsf{r}}, \mathsf{f}, \bar{\mathsf{s}}) = \begin{cases} \overline{\sigma}[(\bar{\mathsf{r}}, \mathsf{f}) \mapsto \overline{\sigma}(\bar{\mathsf{r}}, \mathsf{f}) + 1 * \bar{\mathsf{s}}] & \text{if } \overline{\mathit{isSummary}}(\bar{\mathsf{r}}) = \mathsf{false} \\ \overline{\sigma} & \text{otherwise} \end{cases} \\
\overline{\mathit{inh}} : (\overline{\mathsf{PL}} \times \wp(\overline{\mathsf{L}} \times \mathsf{F} \times \overline{\mathsf{SV}})) \rightarrow \overline{\mathsf{PL}} \\
\overline{\mathit{inh}}(\overline{\sigma}, \{(\bar{\mathsf{r}}_1, \mathsf{f}_1, \bar{\mathsf{s}}_1), \cdots, (\bar{\mathsf{r}}_i, \mathsf{f_i}, \bar{\mathsf{s}}_i)\}) = \overline{\sigma}_i : \\
\qquad\qquad\qquad\qquad \exists \overline{\sigma}_0, \cdots, \overline{\sigma}_i \in \overline{\mathsf{PL}} : \overline{\sigma}_0 = \overline{\sigma} \wedge \forall \mathsf{j} \in [1..\mathsf{i}] : \overline{\sigma}_j = \overline{\mathit{inhS}}(\overline{\sigma}_{j-1}, \bar{\mathsf{r}}_j, \mathsf{f_j}, \bar{\mathsf{s}}_j)
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathit{exhS}} : (\overline{\mathsf{PL}} \times \overline{\mathsf{L}} \times \mathsf{F} \times \overline{\mathsf{SV}}) \rightarrow \overline{\mathsf{PL}} \\
\overline{\mathit{exhS}}(\overline{\sigma}, \bar{\mathsf{r}}, \mathsf{f}, \bar{\mathsf{s}}) = \overline{\sigma}[(\bar{\mathsf{r}}, \mathsf{f}) \mapsto \overline{\sigma}(\bar{\mathsf{r}}, \mathsf{f}) - 1 * \bar{\mathsf{s}}]
\end{array}
$$

$$
\begin{array}{l}
\overline{\mathit{exh}} : (\overline{\mathsf{PL}} \times \wp(\overline{\mathsf{L}} \times \mathsf{F} \times \overline{\mathsf{SV}})) \rightarrow \overline{\mathsf{PL}} \\
\overline{\mathit{exh}}(\overline{\sigma}, \{(\bar{\mathsf{r}}_1, \mathsf{f}_1, \bar{\mathsf{s}}_1), \cdots, (\bar{\mathsf{r}}_i, \mathsf{f_i}, \bar{\mathsf{s}}_i)\}) = \overline{\sigma}_i : \\
\qquad\qquad\qquad\qquad \exists \overline{\sigma}_0, \cdots, \overline{\sigma}_i \in \overline{\mathsf{PL}} : \overline{\sigma}_0 = \overline{\sigma} \wedge \forall \mathsf{j} \in [1..\mathsf{i}] : \overline{\sigma}_j = \overline{\mathit{exhS}}(\overline{\sigma}_{j-1}, \bar{\mathsf{r}}_j, \mathsf{f_j}, \bar{\mathsf{s}}_j)
\end{array}
$$

Fig. 4. Helper functions for the abstract semantics. The prefix operator $\downarrow_1$ denotes the projection of a pair on its first component; it is lifted to sets of pairs.

Finally, we define two functions $\overline{inhS}$ and $\overline{exhS}$ to inhale and exhale permissions, respectively. They map a state of the abstract domain to another state. The permissions are determined by pairs of abstract locations and symbolic values. The functions $\overline{inh}$ and $\overline{exh}$ lift $\overline{inhS}$ and $\overline{exhS}$ to sets of pairs. In order to be sound, we inhale a permission iff the abstract node is not summary. The abstract semantics of statements (Fig. 5) maps a statement, a state of the abstract domain, and a heap to another state. It reflects the permission transfer described in Sec. 2. For instance, acquiring a monitor inhales all the symbolic permissions that its invariant could potentially specify. These are permissions for all abstract locations reachable from the object whose monitor is being acquired. To determine these abstract locations, we apply $\overline{rep}$ to the result of $\overline{reach}$.

Running Example. In method Inc of class W1, we obtain that between the acquire and the release statements (lines 7 and 8), the current thread has the symbolic access permission $1{*}\mathit{Pre}(\mathsf{W1}, \mathsf{Inc}, \mathsf{this.c.f}){+}1{*}\mathit{MI}(\mathsf{Cell}, \mathsf{this.f})$ for each field f of Cell (that is, x, c1, and c2). At the end of the method, it has only $1{*}\mathit{Pre}(\mathsf{W1}, \mathsf{Inc}, \mathsf{this.c.f})$ since we released the monitor of c. The permissions for class W2 are analogous. Before the fork in method main of class OwickiGries, the current thread has $-1{*}\mathit{MI}(\mathsf{Cell}, \mathsf{c.f}){+}\mathsf{full}$ for all fields f of class Cell. The negated permissions from the monitor invariant stem from exhaling the monitor invariant when sharing c; the constant full is inhaled when c is created. When forking the two threads (lines 35 and 36), we exhale the preconditions of the forked methods, obtaining $-1{*}\mathit{MI}(\mathsf{Cell}, \mathsf{c.f}){-}1{*}\mathit{Pre}(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.f}){-}1{*}\mathit{Pre}(\mathsf{W2}, \mathsf{Inc}, \mathsf{c.f}){+}\mathsf{full}$. When joining the forked threads, we inhale the postconditions of the forked methods, and when acquiring c's monitor (line 39), we inhale the monitor invariant of class Cell. Then at line 41, the current thread has $-1{*}\mathit{Pre}(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.f}){+}1{*}\mathit{Post}(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.f}){-}1{*}\mathit{Pre}(\mathsf{W2}, \mathsf{Inc}, \mathsf{c.f}){+}1{*}\mathit{Post}(\mathsf{W2}, \mathsf{Inc}, \mathsf{c.f}){+}\mathsf{full}$ for each field f of Cell.

4.4 Unsound Approximations

The analysis we described so far is sound, but sometimes too coarse in its treatment of summary nodes. Even with a more precise heap analysis, the inference becomes more practical when it uses two unsound approximations.

$$
\begin{array}{l}
\overline{\mathbb{S}} : (\mathsf{St}, \overline{\mathsf{PL}}, \overline{\mathsf{H}}) \rightarrow \overline{\mathsf{PL}} \\
\overline{\mathbb{S}}(\mathsf{x} := \mathsf{E}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\sigma} \\
\overline{\mathbb{S}}(\mathsf{x.f} := \mathsf{E}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\sigma} \\
\overline{\mathbb{S}}(\mathsf{x} := \mathsf{new\ T}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\sigma}[\overline{\mathsf{r}} \mapsto \mathsf{full} : (\overline{\mathsf{r}}, \mathsf{p}) \in \overline{\mathit{reach1}}(\overline{\mathsf{h}'}, \mathsf{x})] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \overline{\mathsf{h}'} \text{ is the abstract heap obtained after } \mathsf{x} := \mathsf{new\ T} \\
\overline{\mathbb{S}}(\mathsf{x.m}(), \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\sigma}_2 : \overline{\sigma}_1 = \overline{\mathit{exh}}(\sigma, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{Pre}(\mathit{class}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x})), \mathsf{m}, \emptyset))) \wedge \\
\qquad\qquad\qquad\qquad\quad \overline{\sigma}_2 = \overline{\mathit{inh}}(\overline{\sigma}_1, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{Post}(\mathit{class}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x})), \mathsf{m}, \emptyset))) \\
\overline{\mathbb{S}}(\mathsf{acquire\ x}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\mathit{inh}}(\sigma, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{MI}(\mathit{class}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x})), \emptyset))) \\
\overline{\mathbb{S}}(\mathsf{release\ x}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\mathit{exh}}(\sigma, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{MI}(\mathit{class}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x})), \emptyset))) \\
\overline{\mathbb{S}}(\mathsf{t} := \mathsf{fork\ x.m}(), \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\mathit{exh}}(\sigma, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{Pre}(\mathit{class}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x})), \mathsf{m}, \emptyset))) \\
\overline{\mathbb{S}}(\mathsf{join\ t}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\mathit{inh}}(\sigma, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{Post}(\mathsf{C}, \mathsf{m}, \emptyset))) : \mathit{TM}(\mathsf{t}) = \mathsf{C.m} \\
\overline{\mathbb{S}}(\mathsf{share\ x}, \overline{\sigma}, \overline{\mathsf{h}}) = \overline{\mathit{exh}}(\sigma, \overline{\mathit{rep}}(\overline{\mathit{reach}}(\overline{\mathbb{E}}(\overline{\mathsf{h}}, \mathsf{x}), \overline{\mathsf{h}}, \emptyset, \mathsf{this}), \mathit{MI}(\mathit{class}(\overline{\mathsf{h}}(\mathsf{x})), \emptyset)))
\end{array}
$$

Fig. 5. The definition of the abstract semantics. The function *TM* yields the method with which a given thread was forked.

Table 2. Instances of permission systems

System	zero	full	fractional	infinitesimal	ensureRead(p)
Fractional	0	1	true	false	$\mathtt{p} > 0$
Counting	0	Integer.MAX_VALUE	false	false	$\mathtt{p} \geq 1$
Chalice	0	100	true	true	$\mathtt{p} \geq \epsilon$

First, as we explained earlier, a sound analysis must not inhale permissions on summary nodes because this might forge permissions. Removing this restriction improves especially the treatment of recursive data structures, which are usually abstracted to summary nodes. Second, our analysis conservatively assumes maximum aliasing in the input state of a method, that is, arguments or fields whose types do not rule out aliasing are represented by summary nodes. Following Clousot [17], we suggest to assume that aliasing does not occur in the input state. This unsound assumption is useful when methods take several parameters of the same type and when a parameter is a recursive data structure.

These unsound approximations may lead to permission annotations that cause a subsequent verification attempt to fail. For instance, unsoundly inhaling on a summary node representing x and y might provide permission to access x.f even if in the concrete execution, there is only permission for y.f. However, in our experiments (see Sec. 6), the unsound approximations helped inferring complete annotations, without compromising their precision.

5 Annotation Inference

In this section, we explain how we infer permission annotations by generating constraints on symbolic permissions and how we solve the constraint system.

5.1 Permission Systems

To support various permission systems, our analysis is parametric in the following aspects: (1) the numerical values that represent permissions, (2) the value that represents the absence of a permission, (3) the value that represents a full permission, and (4) the condition that permits read access. Aspect (1) is expressed via two boolean flags fractional and infinitesimal, which express whether fractional and infinitesimal (ϵ) permissions are supported. Aspects (2) and (3) are expressed via the constants zero and full as presented in the previous section. Aspect (4) is expressed by a function ensureRead : $\overline{\mathsf{PL}} \rightarrow \mathsf{Constr}$ (where Constr is the set of linear constraints over permissions in $\overline{\mathsf{AV}}$). These parameters are aimed at soundly overapproximating different permission systems in a finite way. Therefore they do not define the semantics of concrete systems, but they propose a way of abstracting them. Table 2 presents the parameters for fractional, counting, and Chalice permissions.

Fractional permissions are represented by fractions between 0 and 1, infinitesimal values are not supported, and reading is permitted by any non-zero permission. Counting permissions are represented by integers between 0 and the maximum integer value; again, infinitesimal values are not supported, and reading is permitted by any non-zero permission. We interpret a value i between 0 and $\mathsf{Integer.MAX\_VALUE}/2$ as i counting permissions and a value between $\mathsf{Integer.MAX\_VALUE}/2$ and $\mathsf{Integer.MAX\_VALUE}$ as a full permission minus i counting permissions. Chalice permissions are represented by integers between 0 and 100, infinitesimal values are supported, and reading is permitted by permissions that are at least one infinitesimal permission (symbolic value ϵ).

5.2 Inferring Constraints

A permission-based verification technique prescribes rules that guard the access of heap locations, for instance, that a full permission is required to update the location. We reflect these rules in the analysis through the following constraints on the symbolic permission $\bar{\mathsf{l}}$ for an abstract location at a given program point: (1) $\mathsf{ensureRead}(\bar{\mathsf{l}})$ when the location is read, (2) $\bar{\mathsf{l}} == \mathsf{full}$ when the location is written, (3) $\bar{\mathsf{l}} \leq \mathsf{full}$ after a permission gets inhaled to encode that a method cannot obtain more than a full permission, and (4) $\mathsf{zero} \leq \bar{\mathsf{l}}$ after a permission is exhaled to encode the check that a method must possess the permissions it exhales. To ensure that specifications are self-framing (see Sec. 1), we generate constraint (1) also for field accesses within preconditions, postconditions, and monitor invariants. An additional constraint ensures that all symbolic values represent valid permissions: $\forall \bar{\mathsf{s}} \in \overline{\mathsf{SV}} : \mathsf{zero} \leq \bar{\mathsf{s}} \leq \mathsf{full}$.

In systems that support infinitesimal permissions, we introduce the following constraint on the concrete value of ϵ: $0 < n * \epsilon < 0.5$, where n is the maximal coefficient multiplied by infinitesimal permissions in all symbolic permissions. We interpret permission values in the open interval $(0; 0.5)$ as a positive number of ϵ's and values in $(0.5; 1)$ as 1 plus a negative number of ϵ's.

To infer strong postconditions, we introduce additional constraints for the exit states of the analysis that ensure that each method returns as many permissions to its caller as possible. For each field of a non-summary node reachable through a path p, we determine the upper bound $\bar{\mathsf{l}}$ of the symbolic permissions for all possible exit states of a method m of class C and require $\mathit{Post}(\mathsf{C}, \mathsf{m}, \mathsf{p}) = \bar{\mathsf{l}}$.

Running Example. Fig. 6 reports some of the constraints for the example from Fig. 2. We have already discussed the results of its abstract semantics in Sec. 4.3. For each constraint, we report the code line that induced the constraint. As before, f stands for any field of class Cell (x, c1, or c2).

The first four constraints are introduced for method Inc of class W1. The constraints for class W2 are analogous (with c2 instead of c1). The constraint for line 4 is introduced because Inc's postcondition reads c.c1; in the exit state of the method, the only permission for c.c1 is the one specified in the precondition since we already released the monitor of c. The identical constraint is introduced for the field read **old**(c.c1), which reads c.c1's pre-state value. The field writes

Constraint	Line
ensureRead($1*\mathit{Post}$(W1, Inc, c.c1))	4
$1*\mathit{Pre}$(W1, Inc, c.x) + $1*\mathit{MI}$(Cell, x) = full	7
$1*\mathit{Pre}$(W1, Inc, c.c1) + $1*\mathit{MI}$(Cell, c1) = full	8
$1*\mathit{Post}$(W1, Inc, c.f) = $1*\mathit{Pre}$(W1, Inc, c.f)	10
ensureRead($1*\mathit{MI}$(Cell, f))	14
zero $\leq$ full $- 1 * \mathit{MI}$(Cell, f) $- 1 * \mathit{Pre}$(W1, Inc, c.f) $- 1 * \mathit{Pre}$(W2, Inc, c.f)	36

Fig. 6. Some constraints for the running example

to c.x and c.c1 (lines 7 and 8) require that the method has write permission for the corresponding abstract locations. Therefore, we introduce a constraint that for these locations, the sum of the permissions in Inc's precondition and in Cell's monitor invariant must be a full permission. By the abstract semantics the permissions in the exit state of Inc are exactly those specified in the precondition. So we enforce that the precondition and the postcondition specify the same permissions for each field f of c. The monitor invariant of class Cell (line 14) reads all fields of the class and, thus, requires read permission for them.

Several constraints are produced for the main method of class OwickiGries. We discuss the one for the second fork (line 36). By the heap analysis, we know that c is fresh in method main. So the permission held after the second fork for any field c.f is the full permission (from the creation of c) minus what is specified in Cell's monitor invariant (from sharing c) minus what is specified in W1.Inc's precondition (from the first fork) minus what is specified in W2.Inc's precondition (from the second fork). The constraint ensures that main has sufficient permissions for the second fork, that is, that exhaling the precondition does not lead to permissions smaller than zero.

5.3 Resolution of the Constraints

We solve the inferred constraint system using linear programming [8]. We define an objective function that lets us infer the *minimal* permissions that satisfy the constraints. Maximizing the permissions would often result in full permissions for each reachable location, even if the location is never accessed. Such a solution complicates subsequent verification, for instance, by providing weaker framing.

Through the objective function we also express *priorities* defining where to put annotations when several solutions are possible, for instance, in the method specification or in the monitor invariant. To do that, we multiply each symbolic value in the objective function by a factor. A bigger factor expresses a lower priority for that symbolic value, since we minimize the objective function.

Solving the linear programming system determines whether the system is feasible, that is, whether there are numerical values (real numbers) for all symbolic values that satisfy the constraints. An infeasible system may occur because of approximation, for instance, if we soundly abstain from inhaling on summary nodes, the constraint for a subsequent field access might not be satisfiable. If the system is feasible, we use the solution to compute the permission predicates for pre- and postconditions as well as monitor invariants.

The constraints resolution provides a numerical value that has to be translated to a permission predicate. This step is straightforward for fractional permissions. For counting permissions, we translate a value differently, depending on whether it is less or greater than Integer.MAX_VALUE/2 (see Sec. 5.1). For Chalice permissions, the integer part of each numerical value is turned into a percentage, whereas the mantissa is turned into a (positive or negative) number of counting permissions by dividing it by the solution for the symbolic value ϵ.

Running Example. In the following, we present the annotations obtained by solving the constraints in Fig. 6 for Chalice's permission model. Fractional and counting permissions lead to similar results. The constraint system is feasible, and we obtain different solutions, depending on the priorities encoded in the objective function. Here, we give priority to monitor invariants. Assume that the numerical value for ϵ is 0.1. Then for W1's Inc method we obtain 0.1 for c.c1 and 0 for all other fields, which are the smallest possible values that satisfy the constraints (especially the first) in Fig. 6. This solution results in acc(c.c1, ϵ) for the pre- and postcondition, and analogous results for W2.Inc.

By the second and third constraint, and by the analogous constraints for W2.Inc, we obtain for Cell's monitor invariant 99.9 (that is, $100-\epsilon$) permission for c.c1 and c.c2, and 100 for c.x. This solution cannot be expressed in Chalice, which does not have syntax for a negative number of ϵ's. However, we could easily add a constraint that for each symbolic value, the mantissa of the numerical value in the solution must be in $[0; 0.5)$ and, thus, translate into a non-negative number of ϵ's. With this additional constraint, we obtain the pre- and postcondition **acc**(c.c1,1), and the monitor invariant **acc**(x) && **acc**(c1,99) && **acc**(c2,99). This solution reflects the need to split the permissions as discussed in Sec. 2, and allows one to verify the example in Chalice.

6 Experimental Results

We implemented our inference system in Sample, a generic compositional static analyzer. We executed the analysis on an Intel Code 2 Quad CPU 2.83 GHz with 4 GB of RAM, running Windows 7, and the Java SE Runtime Environment 1.6.0_16-b01. Table 3 summarizes the experimental results when we apply the analysis to case studies taken from (i) the Chalice tutorial [15] and the Chalice distribution, (ii) VeriCool [24], and (iii) VeriFast [12] libraries. Sample analyzes Scala programs. Therefore all the examples have been written in Scala using a custom library to represent statements that are not natively supported.

We performed the experiments applying the heap analysis with the unsound entry state and unsound inhaling, giving higher priorities to monitor invariants. Column Program reports the program we analyzed and LOC the lines of code; columns Fractional, Counting, and Chalice report the time of the analysis (in msec) when using fractional, counting, and Chalice permissions, respectively. % Inferred Contracts reports the percentage of inferred contracts including

Table 3. Experimental results

Program	LOC	Fractional	Counting	Chalice	% Inferred Contracts	Heap Analysis
Fig1	20	45	50	55	100%	22
Fig2	12	12	9	8	100%	11
Fig3	13	9	6	7	100%	8
Fig4	25	3	3	5	100%	8
Fig5	24	143	142	163	100%	80
Fig6	27	53	50	61	100%	20
Fig11	32	15	9	17	100%	20
Fig12	31	15	13	23	100%	25
Fig13	35	706	726	760	100%	223
OwickiGries	59	164	129	131	100%	39
cell − defaults	164	115	97	120	100%	55
linkedlist	77	78	82	86	100%	61
swap	15	10	9	10	100%	5
AssociationList	113	668	753	741	36%	305
HandOverHand	128	564	532	611	36%	478
Master	65	76	81	89	100%	57
CellLib	116	148	154	160	100%	79
CompositePattern	67	1217	1282	1279	71%	1009
Spouse	58	221	135	164	100%	33
Account	52	12	9	9	100%	16
Stack	54	76	74	78	67%	35
Iterator	57	46	55	53	100%	28

loop invariants w.r.t. the contracts that were in the original annotated program. Column `Heap Analysis` contains the time of the heap analysis (in msec).

The analysis takes less than a second in all cases except CompositePattern, and the times of execution are similar using different permission systems. We were able to infer all contracts for most of the examples, obtaining the same precision using different permission systems. On the other hand, we infer only one third of the annotation for AssociationList and HandOverHand and two thirds for Stack since these examples deal with recursive data structures, which are roughly approximated by our heap analysis. Similarly, CompositePattern contains a set of nodes that is roughly abstracted by the heap analysis, and so our approach is able to infer annotations for the fields of the class but not for the elements contained in such a set. The verification of programs with partial annotations would fail, but the user could manually add the missing contracts.

7 Related Work

There is a large body of work on the inference of program annotations. Ernst et al.'s Daikon system [9] uses a dynamic analysis to infer object invariants. Flanagan and Leino's Houdini tool [11] generates a large number of candidate annotations and uses ESC/Java to verify or refute each of them. Leino and Logozzo [13] integrate abstract interpretation and program verification to infer loop invariants. However, none of these inferences supports access permissions.

The Chalice language [14] provides an option `-autoMagic` to infer certain permission predicates. However, the inference does not find non-trivial splittings of permissions as required by our running example, and it cannot be applied to other permission models.

Calcagno et al. [3] propose an inference system based on bi-abduction. Their approach uses a compositional shape analysis to infer annotations over separation logic formulas. The approach has been extended to infer resource invariants for concurrent programs [4], that is, the (full) access permissions that are associated with a lock.

Yasuoka and Terauchi [27] propose a calculus to infer fractional permissions. Like our approach, they represent constraints with linear inequalities, and they solve them using linear programming. Their approach is focused on a simple region language, and it does not support object-oriented features and concurrency.

A major application of access permissions is to simplify framing, that is, determining what is definitely not changed by a method execution. There are several static analyses for frame information. Rakamarić and Hu [21] propose a technique to infer frame information for functions and loops on C programs. Spoto and Poll [26] introduce a static analysis based on abstract interpretation for JML's assignable clauses. However, their analysis only checks existing annotations, rather than inferring annotations. Cataño and Huisman's Chase tool [5] performs similar checks. The practical effectiveness of their approach has been demonstrated both in terms of precision and efficiency on industrial code. However, the approach is not sound, since it does not consider aliasing. In contrast to these approaches, we infer access permissions, which can then be used to infer framing information [25], and for other purposes like verifying concurrent code.

8 Conclusion

We presented an analysis to infer access permissions for various permission systems. Our approach infers pre- and postconditions and monitor invariants. It also handles loop invariants and abstract predicates, but we omitted them in the paper for brevity. The experimental results indicate that our analysis is efficient and precise. As future work, we plan to increase the precision of our approach adopting shape analysis [16] to obtain more precise heap abstractions, and to mutually refine the heap abstraction and the permission inference through a reduced product [7]. We also plan to extend the analysis to permission predicates where the permission is expressed by a program variable.

Acknowledgments. We are grateful to the anonymous referees, Agostino Cortesi, and Alexander J. Summers for their helpful feedback. This work was partially supported by the SNF project "Verification-Driven Inference of Contracts".

References

1. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: POPL 2005. ACM (2005)
2. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)

3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL 2009. ACM (2009)
4. Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive Resource Invariant Synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 259–274. Springer, Heidelberg (2009)
5. Cataño, N., Huisman, M.: CHASE: A Static Checker for JML's *Assignable* Clause. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 26–40. Springer, Heidelberg (2002)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977. ACM (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979. ACM (1979)
8. Dantzig, G.B.: Linear programming and extensions. Rand Corporation Research Study. Princeton Univ. Press (1963)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., Mccamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69, 35–45 (2007)
10. Ferrara, P.: A fast and precise analysis for data race detection. In: Bytecode (2008)
11. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
13. Leino, K.R.M., Logozzo, F.: Loop Invariants on Demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
14. Leino, K.R.M., Müller, P.: A Basis for Verifying Multi-threaded Programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
15. Leino, K.R.M., Müller, P., Smans, J.: Verification of Concurrent Programs with Chalice. In: FOSAD 2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
16. Lev-Ami, T., Sagiv, M.: TVLA: A System for Implementing Static Analyses. In: SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)
17. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
18. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
19. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM 19, 279–285 (1976)
20. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL 2005. ACM (2005)
21. Rakamaric, Z., Hu, A.J.: Automatic inference of frame axioms using static analysis. In: ASE 2008. IEEE (2008)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE Computer Society (2002)

23. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM ToPLaS 24(3), 217–298 (2002)
24. Smans, J., Jacobs, B., Piessens, F.: VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 220–239. Springer, Heidelberg (2008)
25. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
26. Spoto, F., Poll, E.: Static analysis for JML's assignable clauses. In: FOOL 2003 (2003)
27. Yasuoka, H., Terauchi, T.: Polymorphic Fractional Capabilities. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 36–51. Springer, Heidelberg (2009)