Componentized Heap Abstraction

Technical Report TAU-CS-164/06

N. Rinetzky G. Ramalingam E. Yahav M. Sagiv

Abstract

In this paper we present a new heap abstraction that seeks to strike a balance between the use of non-local (transitive) properties to gain precision and exploiting heap-locality. The abstraction represents the heap as an (evolving) tree of heap-components, with only a single heap-component being accessible at any time. The representation is tailored to yield several benefits: (a) It localizes the effect of heap mutation, enabling more efficient processing of heap mutations; (b) The representation is more space-efficient as it permits heap-components with isomorphic contents to use a shared representation; (c) It enables a more precise identification of the "input heap" to a procedure, increasing the reuse of summaries in a tabulation-based interprocedural analysis, making it more efficient. Furthermore, based on our new abstraction, an analysis can compute parameterized summaries which can be re-used for analyzing clients of instantiations of the generic data-structures.

1 Introduction

Dynamically-allocated data structures are ubiquitous in today's software systems. Static analyses of programs, hence, typically rely on various techniques to create a finite abstraction of a potentially unbounded heap. Existing shape analysis techniques [6, 10–12, 18, 20, 21, 28, 30, 32, 33] have been quite successful in various complex examples, such as verifying that a procedure correctly sorts a list implemented using a linked data structure. This paper suggests a novel heap abstraction framework that we hope will make these techniques more robust, with regards to both precision and efficiency.

One of the main goals of this work is to exploit "heap locality": the heap usually consists of many "parts" and at any point during program execution only some of these "parts" may be relevant. However, the abstractions used in existing shape analyses, e.g., [32], depend significantly on the use of *transitive properties*, such as reachability information: *e.g.*, the abstraction may distinguish between two heap objects because one is *transitively reachable* from a variable x, while the other is not, even though the objects may be otherwise identical. These properties help maintain necessary distinctions in the abstraction, especially in the middle of a complex sequence of pointer operations implementing a data structure transformation.

A straightforward use of such transitive properties can, however, lead to analyses and abstractions that are "non-local", with some disadvantages that we will describe soon. The abstraction we present in this paper attempts to balance the use of such transitive properties with the goal of exploiting locality in the abstraction.

Technically, we use a non-standard operational semantics which represents the heap as a tree of components: The only inter-component pointer references allowed are from an object in a parent component to an object in a child component. At any point during execution, only a single component (the *current* component) is accessible (and relevant). The abstraction can track more information for the current component than other components. Transitive properties are captured only within components and possibly in the representation of the tree of components.

This representation has three main benefits: (i) It localizes the effect of heap mutations. In particular, after a mutation, reachability information is updated only for the current component, and by examining only the current component. (ii) It makes the representation more space-efficient, by allowing components with "identical" contents to use a shared representation. Such sharing is made more likely by lifting transitive properties like reachability from variables out of the representation of a component's content. (iii) As a consequence of the above, it is possible to more precisely identify the "input heap" to a procedure. This makes tabulation-based interprocedural analysis cheaper since smaller summary tables are likely. Utilizing an abstraction of our heap representation, an analysis can compute parameterized summaries which can be re-used for analyzing clients of instantiations of the generic data-structures.

Currently, our abstraction relies on hints from programmers to identify the "tree of components".

1.1 A Running Example

Fig. 1 presents a Java 5.0 program which we will use to illustrate our ideas. This program generates a set of tasks, stores them in a (round-robin) queue of pending tasks, and iteratively processes these tasks. When a task is completed, it is moved to the completed-queue. Every task contains a queue of subtasks. Every time a task is processed, some of its subtasks are executed. Finished subtasks are returned to the (tail of) the subtasks-queue. A finished subtask at the head of the subtasks-queue indicates that the task is completed. The code in comments (referring to seal and unseal operations) will be explained later.

Our goal is to verify that all generated tasks have been completed and reside in the completed-queue when the program terminates. Our analysis utilizes the fact that the tasks stored in the work-queue are not modified by the execution of the generic queue methods and that `iterate` extracts a task from the work-queue before processing it.

1.2 The Standard Heap Representation

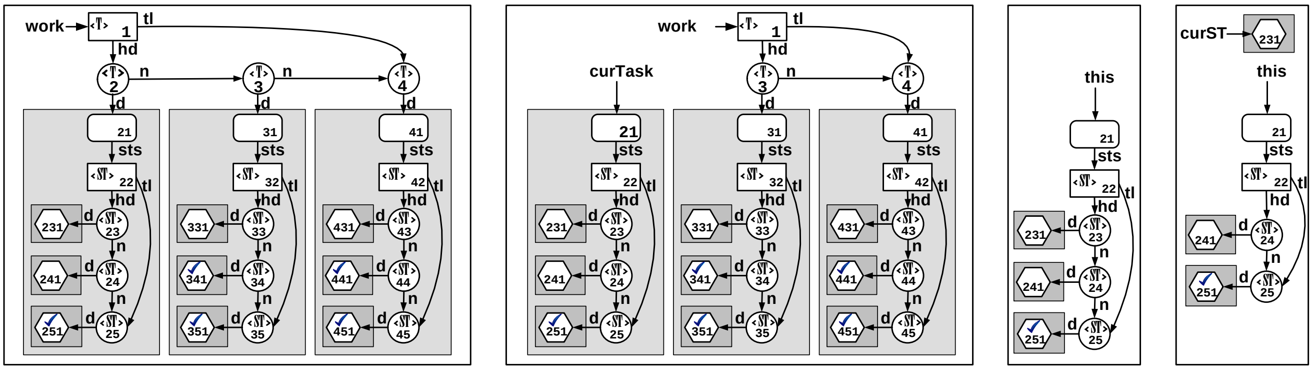
Fig. 2 depicts a standard *monolithic* representation of memory states that arise at the call-site and return-site of the generic `deq` method. Objects are depicted as nodes. The shape of a node encodes the type of the object it depicts: rounded-corner rectangles and hexagons depict `Task` and `SubTask` objects, respectively; rectangles and circles annotated by `<E>` depict instantiations of the generic `GQueue` and `GQueueNode` parameterized with type parameter `<E>`. (`<T>` stands `<Task>` and `<ST>` for `<SubTask>`.)

```
public class Main {
public static void main(String argv[]) {
  GQueue⟨Task⟩ pending = new GQueue⟨Task⟩();
  GQueue⟨Task⟩ completed = new GQueue⟨Task⟩();
  genTasks(pending);

  while (!pending.isEmpty())
    Task did = iterate(pending)
    completed.enq(did);
  }
}
. . .
private static Task iterate(GQueue⟨Task⟩ work)
  while (true) {
    Task curTask = work.deq();
    // unseal(curTask);
    if(curTask.process())
      return curTask;
    else
      // seal(curTask);
      work.enq(curTask);
  }
}
}
```

```
public class Task {
private GQueue⟨SubTask⟩ sts; . . .
public boolean process() {
  boolean taskDone = false;
  while (... && !taskDone) {
    SubTask curST = this.sts.deq();
    // unseal(curST);
    if (curST.isDone())
      taskDone = true;
    else
      curST.execute();
    // seal(curST);
    this.sts.enq(curST);
  }
  return taskDone;
} . . .
}

public class SubTask {
private boolean isDone = false;
public void execute() { . . .
  this.isDone = true;
}
}
```

```
public class GQueue⟨E⟩ {
GQueueNode⟨E⟩ hd, tl;
. . .
public E deq() {
  if (this.hd == null)
    throw new InternalError();

  E ret = this.hd.d;
  this.hd = this.hd.n;
  if (this.hd == null)
    this.tl = null;
  return ret;
} . . .
}


public class GQueueNode⟨E⟩ {
  GQueueNode⟨E⟩ n;
  E d;
}
```

Figure 1: The running example. The omitted code appears in App. A.



(s_c^T) (s_r^T) (s_c^{ST}) (s_r^{ST})

Figure 2: Standard memory states occurring during execution of the running example.

The values of a pointer field `f` are depicted as f-labelled edges. The value of a pointer variable `x` is depicted as an edge emanating from the label x. A checkmark in a hexagon indicates that a subtask is finished (i.e., its `done` field is true). The numbers attached to nodes indicate the *location* of objects.

The shaded rectangles in Fig. 2 correspond to a high-level view of the different *components* making up the heap. A programmer may consider the `work` queue in s_c^T as a component with 3 subcomponents (*i.e.*, queue with 3 task elements). Similarly, she may think of each task as a component consisting of 3 subtask elements (subcomponents).

This high-level, hierarchical and modular, view of the heap is not explicitly captured in the standard heap representation. Existing shape analyses [30, 31] make use of a *monolithic and flat* abstract heap representation that also fails to capture this hierarchical and modular view. Such an abstraction has certain disadvantages. For example, this reduces the possible reuse of procedure summaries in the analysis: s_c^T and s_c^{ST} depict memory states that occurs in method `iterate` and `process` in which the (generic) `deq` is invoked, respectively. In both states the two queues contain 3 elements. Though the procedure's behavior is similar in both cases, the abstract inputs to the procedure in the two cases look different. As a result, the procedure summary computed for one of these inputs cannot be reused for the other input. However, a more modular heap abstraction can capture the similarity of the inputs to the procedure in the two cases and enable reuse of the procedure summary.

1.3 An Overview of Our Approach

We now informally introduce the basic ideas underlying our work. We start off with a non-standard semantics, $\mathcal{HH}$, that explicitly maintains the component-decomposition of the heap as a tree of components. We then define a finite abstraction of this non-standard heap representation in a fairly natural, compositional, way.

1.3.1 A Non-Standard Concrete Heap Representation

Every shaded rectangle in s_c^T, excluding any nested rectangle, can be considered as a component. The nesting of the shaded rectangles depicts the parent-child relationship between components. For example, in state s_c^T, the topmost component contains the queue object pointed-to by variable `work` and the 3 queue nodes. This component has 3 *child* components. Each child component is a task, consisting of a task object containing a queue with 3 subtasks. Each subtask is a (trivial) component of its own. We refer to the top-most component as the *current component* and to all other components as *sealed* components.

Our non-standard concrete semantics represents the heap as a tree of components (referred to as a *macroheap*). The internal structure of a component is described by a *microheap*, which looks like a conventional (standard) heap. However, note that the microheap representation itself does not explicitly capture any pointer references between objects belonging to different microheaps. Instead, such pointer references are captured as tree edges in the component-decomposition tree (or macroheap). This allows reuse of the same microheap to describe multiple components with isomorphic internal structure. For example, the 3 tasks components shown in s_c^T can be represented by the same microheap μh consisting of a task which has 3 subtasks (even though the subtasks are different for each task).

1.3.2 Component Decomposition

The component-decomposition is modified by the execution of `seal` and `unseal` statements. `seal(x)` seals the subheap reachable from `x` into a new component which becomes a subcomponent of the current component. The program is allowed to access only the contents of the current component. In order to access the contents of a sealed component, an `unseal` command needs to be executed. `unseal(x)` unseals the component pointed-to by `x`, making it part of the current component.

We assume that seal/unseal directives are provided as part of the input program. Heuristics can be used to automatically perform seal/unseal operations in certain contexts (such as procedure calls), as discussed later.

1.3.3 An Abstract Heap Representation

We abstract the whole heap by first abstracting the microheaps (using an abstraction similar to the canonical abstraction [32] used in previous shape analyses), and then abstracting the component-decomposition tree into an abstract component-decomposition graph by merging components with identical abstractions. A more precise abstraction can be used for the current component than for the other components. The abstraction also permits capturing "non-local" properties in a controlled fashion (*e.g.*, to permit the abstraction of a component to depend on information from its child components), details of which are presented in Section 3.

1.3.4 Ignoring Irrelevant Subtrees

Existing tabulation-based interprocedural analyses [6, 18, 30, 31] increase the reuse of procedure summaries by analyzing procedure calls only on the subheap reachable from the procedure's parameters (referred to as the procedure's *local*-heap). Thus, the subheap reachable from the procedure's parameters is used as a conservative approximation for the part of the heap that is relevant to the procedure call (*i.e.*, its "input heap").

Usually, not all of the subheap reachable from a procedure's parameters is actually accessed by the procedure. The component-tree representation is designed to allow execution using an "incomplete" component-tree, *i.e.*, a component-tree where certain subtrees may be missing. This permits the analysis to analyze procedures on memory states which contain only a subset of their local heap.

Given a specification of which components are (ir)relevant for a procedure p, a procedure invocation proceeds by "cutting away" the contents of the irrelevant components. Technically, this is done by replacing these irrelevant components by *opaque* components (*i.e.* components with no content). When p executes, it is allowed to manipulate references to the opaque components, but not to inspect their contents.(One may think of opaque components as "named holes" in the heap.) When the procedure returns the (unmodified) contents of the opaque components are pasted back in place.

For example, if the subcomponents stored in the queue are specified to be irrelevant for `deq`, then invoking `work.deq()` and `this.sts.deq()` on memory states s_c^T and s_c^{ST}, respectively, will result in executing `deq` on the same memory state: a queue containing 3 opaque components. The reduced procedure local-heaps increase possibilities for reusing procedure summaries in an abstract interpretation of $\mathcal{HH}$.

1.3.5 Generics

To enjoy the benefits of a shared representation of parts of the heap, the programmer needs to specify when to seal and when to unseal components. To benefit from an increased summary reuse, the programmer also needs to specify which components are (ir)relevant to a method.

For programs using generic implementations of data structures we provide an automatic derivation for the above specification. Basically, every parameter to a generic method which is of the parametric type is sealed in an opaque component before the method is invoked, and every time a method returns an object of the parametric type, this object is treated as an opaque component and is unsealed.

1.4 Limitations and Simplifying Assumptions

1.4.1 Limitations

The main limitations of our approach are (a) it requires the component-decomposition to form a tree, and (b) the only inter-component pointer reference allowed are from a parent component to a child component. We also require that a component have only one in-port (an in-port is an object that is a target of an inter-component pointer reference). This is, however, not a real restriction. In App. D, we show how multiple in-ports can be emulated using a single in-port. However, multiple in-ports make the analysis more complicated (and possibly less precise).

Our approach relies on user specification of the component-tree. The automatic approach we provide for generic data structure does not allow analysis of arbitrary methods, e.g., ones that mutate their elements.

1.4.2 Simplifying Assumptions

We assume that in every procedure call, the actual parameters dominate the subheap reachable from them. This restriction, called cutpoint-freedom in [31], simplifies the analysis. We note that our semantics can be easily generalized to handle programs with arbitrary procedure calls. However, computing procedure summaries that can be used in such a case is a challenging, but orthogonal, problem to the one we addressed here. (For additional information, see [30]). We also make the simplifying assumption that the program does not have global variables and that formal parameters are not modified.

2 Hierarchical-Heaps Concrete Semantics

In this section, we describe the $\mathcal{HH}$ semantics. $\mathcal{HH}$ is a natural (large-step) semantics (see, *e.g.*, [9]). For brevity, we only discuss the key aspects of the operational semantics, formally defined in App. B.

Domain			Description
l	$\in$	Loc	Locations
v	$\in$	$Val = Loc \cup \{null\} \cup \{true, false\}$	Values
f	$\in$	$\mathcal{F}$	Fields names
$type$	$\in$	$\mathcal{T}$	Types names
c	$\in$	$\mathcal{C}$	Components identifiers
ip	$\in$	$\mathcal{IP} = \{\{0, \ldots, n\} \to Loc \mid n \in \mathcal{N}\}$	In-ports
h	$\in$	$\mathcal{H} = Loc \rightharpoonup \mathcal{F} \rightharpoonup Val$	Intra-microheap link structure
t	$\in$	$\mathcal{T} = Loc \rightharpoonup \mathcal{T}$	Object-to-type map
μh	$\in$	$\mu\mathcal{H} = \mathcal{IP} \times 2^{Loc} \times 2^{Loc} \times \mathcal{H} \times \mathcal{T}$	Microheaps
e	$\in$	$\mathcal{E} = \mathcal{C} \rightharpoonup Loc \rightharpoonup \mathcal{C}$	Hierarchical backbone
d	$\in$	$\mathcal{D} = \mathcal{C} \rightharpoonup \mu\mathcal{H}$	Content-descriptors
${}_{M}h$	$\in$	${}_{M}\mathcal{H} = \mathcal{C} \times 2^{\mathcal{C}} \times \mathcal{E} \times \mathcal{D}$	Hierarchical (macro) heaps
ρ	$\in$	$Env = VarId \rightharpoonup Val$	Variable environment
δ	$\in$	$\Delta = \mathcal{C} \rightharpoonup \mathcal{C}$	Opaque components renaming map
σ	$\in$	$\Sigma = Env \times {}_{M}\mathcal{H} \times \Delta$	Hierarchical memory states

Figure 3: Semantic domains of the $\mathcal{HH}$ semantics.

2.1 Concrete Memory States

Fig. 3 defines the concrete semantics domains and meta-variables ranging over them.

A *hierarchical memory state* is a triplet $\sigma = \langle \rho, {}_{M}h, \delta \rangle$, where: (i) $\rho \in Env$ is an environment assigning values to variables (since $\mathcal{HH}$ is a natural semantics, ρ assigns values only for the variables of the *current* procedure.); (ii) ${}_{M}h \in {}_{M}\mathcal{H}$ is a hierarchical macroheap; and (iii) $\delta \in \Delta$ is explained in Sec. 2.3.

2.1.1 The Macroheap

The hierarchical macroheap component of the state, ${}_{M}h = \langle c, C, e, d \rangle$ explicitly records a partitioning of the dynamically allocated objects into a set of (disjoint) heap-components. Every component is identified by a unique component-identifier $c' \in C \subseteq \mathcal{C}$. When an object in component c_1 has a field pointing to an object in component c_2 we say that component c_1 *contains* component c_2. The semantics ensures that the containment relation forms a *tree*. The root of the tree is the *current component* $c \in \mathcal{C}$. The *hierarchical backbone* $e \in \mathcal{E}$ records the component-containment relation (i.e., the edges of the component-tree). We refer to the children of a component c_1 in the component-tree as c_1's *subcomponents*.

The partial function $d \in \mathcal{D}$ associates a subset of the components in C with *microheaps* which describes the component's content. A component $c \in C$ is *transparent* when $c \in dom(d)$ and otherwise it is *opaque*. As we shall see, opaque components are leaves in the component-tree.

Example 1 *Fig. 4 shows a* simplified[1] *macroheap*

[1] Some details are omitted for clarity of presentation. Later we show examples with full details.

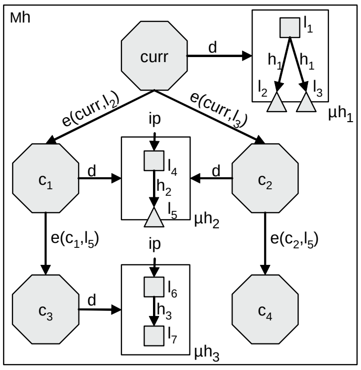


Figure 4: Sample macroheap ${}_{M}h$.

${}_{M}h{=}\langle curr, \{curr, c_1, c_2, c_3, c_4\}, e, d\rangle$*, containing* 5 *components, drawn as octagons. For now, the microheaps, drawn inside the* 3 *rectangles, can be ignored. The current component* $curr$ *is the root of the containment hierarchy. The* d*-labeled edges depict the mapping of components to microheaps.* c_4 *is an opaque component and thus its content is not represented in the memory state. Components in the macroheap are connected by* e*-edges.*

2.1.2 Microheaps

The content of a component is described using a microheap. On its own, a microheap $\mu h \in \langle ip, L^O, L^C, h, t\rangle$ is essentially a 2-level store. The two disjoint sets $L^O \subset Loc$ and $L^C \subset Loc$ comprise the *locations of the microheap*. The set L^O contains locations identifying the microheap's objects. The set L^C contains locations identifying the microheap's child-ports. (We describe the role of child-ports in the next paragraph.) The partial function $h \in \mathcal{H}$ assigns values to fields of objects. A value may be either an atomic value or a location *inside* the microheap. The partial function $t \in \mathcal{T}$ maps every object to its type-identifier.

The externalization of the content of components allows using the same microheap for describing the contents of several components. For example, microheap μh_2 of Fig. 4, represents the content of both components c_1 and c_2. This means that the (disjoint) heap parts comprising c_1 and c_2 have isomorphic internal structure. (For a formal definition, see Fig. 15 in App. B.) It does *not* mean that c_1 and c_2 have shared objects. Thus, microheaps should be thought of as templates, or blueprints, for the content of components and not as subheaps.

2.1.3 The Hierarchical Backbone

The microheap associated with a component c describes only the content of c, but not the contents of c's subcomponents. The hierarchical backbone $e \in \mathcal{E}$ maintains the inter-component link-structure: Every inter-component reference leaving a component

c is interposed with a *child-port*. A child-port is a location. In the microheap associated with c, a child-port represents the target of inter-component references. In the macroheap, a child-port labels an edge from c to a subcomponent.

Technically, the hierarchical backbone e associates every component c_s with a map from the child-ports of the microheap $d(c_s) = \langle ip_s, L_s^O, L_s^C, h_s, t_s \rangle$ associated with c_s, to a subcomponent c_t of c_s. This means, that any pointer to a child-port $l \in L_s^C$ represents a reference to an object in $e(c_s)(l)$. Our semantics allows for only one object inside a component to be referenced from an object outside that component (the *component's header*). Thus, to determine the target of inter-component references, it suffices to know the header of every component. The microheap associated with the component represents the header object by having $ip = 1 \mapsto l_h$, where l_h is the location of the microheap's object representing the header. We refer to l_h as the *microheap's header*.

Example 2 *Fig. 4 shows* 3 *microheaps. The microheap associated with component* c_2 *is* $\mu h_2 = d(c_2) = \langle 1 \mapsto l_4, \{l_4\}, \{l_5\}, h_2, t_2 \rangle$. *This microheap consists of the single object in location* l_4 *and the single child-port location* l_5 *(depicted as a triangle). The object in location* l_4 *is the microheap's header. The heap link-structure* inside *this microheap is captured by the function* h_2. *For simplicity, we do not show either the field names, the function* t_2 *mapping locations to their type, or the environment.*

Components are connected via e*-labeled edges. An* e *edge connects a component to a subcomponent through a child-port of the containing component. The child-ports of a component correspond to locations in its microheap.*

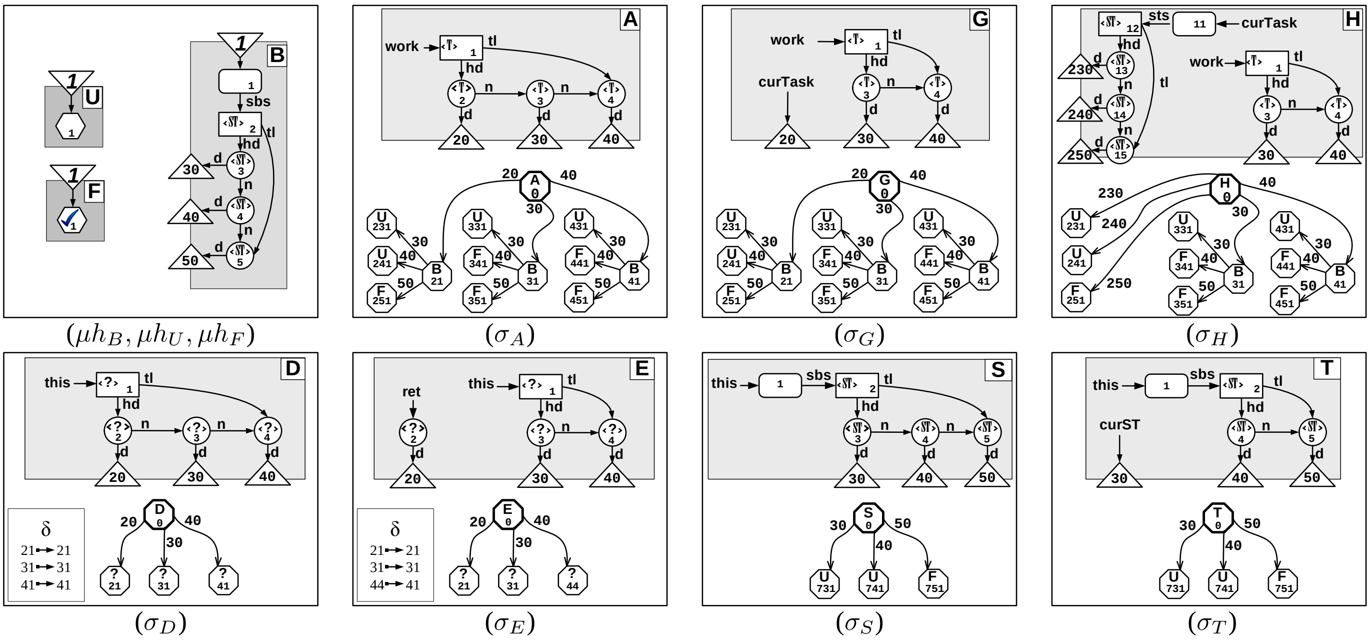
The representation of the targets of inter-component references as locations allows to treat pointer fields inside a microheap in a uniform way. In particular, it allows to mutate fields holding references to subcomponents as well as to perform pointer equality checks using only the information maintained in the microheap, without requiring to consult the macroheap. Locations also provide unique labels for the (unbounded number of) inter-component edges. Furthermore, the same labels can be used for all the components represented by a microheap.

2.1.4 Shared Representation of Memory States

The microheaps used to externalize the representation of components can be shared within, and across, memory states.

Example 3 *Fig. 5 shows* 7 *memory states that arise in the running example. The* 3 *microheaps shown in (*$\mu h_B, \mu h_U, \mu h_F$*) are reused in the representation of* 5 *of the memory states. (We show them separately to simplify presentation.) In this figure we use integer values as locations and component identifiers.*

For every memory state, we show its component-tree and the microheaps corresponding to component contents, possibly referring to microheaps in ($\mu h_B, \mu h_U, \mu h_F$*). The current component is (always) the root of the component tree. Microheaps are shown inside shaded rectangles labeled by a capital letter. We refer to microheaps in the diagram by using these labels, e.g.,* μh_B *is depicted inside the* B*-labeled shaded rectangle. The mapping of a transparent component to its microheap is shown by the*



$(\mu h_B, \mu h_U, \mu h_F)$ (σ_A) (σ_G) (σ_H)

(σ_D) (σ_E) (σ_S) (σ_T)

Figure 5: Hierarchical memory states occurring in the running example.

microheap label appearing inside the component node. For an opaque component, we write $?$ inside the component node to denote that it is not mapped to a microheap.

The header of the microheap is pointed to by an edge from a 1-labeled down-pointing triangle. The linking of a child-port to a subcomponent is depicted by labeling the component-tree edges with the child-port-location.

Note that microheaps provide a shared representation of component contents. For example, all the components containing a task with 3 elements are represented by μh_B, which has 3 child-ports. In state σ_A, which occurs when `curTask=work.deq()` is invoked, the 3 components containing the tasks held inside the work-queue are represented by μh_B. Note that the representation of these components is shared although they are linked to different subcomponents. For example, child-port 40 links every task to its second subtask. In component 21 the second subtask is unfinished, while in components 31 and 41 the second subtask is finished.

Remark. For simplicity, we chose to use a store-based representation of memory states, using explicit values as locations and component identifiers. Alternatively, we could have used a storeless semantics [19], but this further complicates the presentation. We note that our abstraction is indifferent to the actual values of these identifiers.

2.2 Intraprocedural Statements

Intraprocedural statements are handled as usual in a two-level store semantics for pointer programs (e.g., see [29]), using the microheap associated with the current-component, which we refer to as the *current-microheap*, as the program's store. The environment maps pointer variables (of the current procedure) only to locations inside the current-microheap. References to child-ports can be used only as RHS-values (right-hand-side-values).

The programmer can mutate the partitioning of the heap into components using explicit `seal` and `unseal` operations. The `seal` command is used as a guarantee by the programmer that certain parts of the heap will not be accessed until being `unseal`ed.

A `seal(x)` command checks that the subheap reachable from variable `x` is dominated by the object pointed-to by `x`; extracts this subheap from the current-microheap; redirects all the references to the object pointed-to by `x` to a new child-port l_{new} allocated in the current-component; creates a new component c_{new}; makes c_{new} a child of the current-component by linking l_{new} to c, i.e., the macroheap edge between the the current-component and c_{new} is labeled with l_{new}; associates c_{new} to a microheap which can represent the extracted subheap, using an existing microheap if possible.

An `unseal(x)` command checks that `x` points to a child-port l_{old} linked to a transparent subcomponent c_{old} of the current-component; absorbs a copy of the microheap associated with c_{old} into the current-microheap; redirects all the references to l_{old} to the header of the absorbed microheap; and discards of c_{us} and l_{old}.

An attempt to dereference a pointer pointing to a child-port, or a failure in one of the aforementioned checks is a *sealing error* which aborts the program. The information maintained in the current-microheap suffices for performing the check associated with pointer-dereferences as well as the domination check, associated with the `seal` command.

Example 4 *Executing `seal(curTask)` in state σ_H of Fig. 5, results in state σ_G. Note that μh_B is reused to represent the contents of the new component* 21. *Also note that the components that are linked to child-ports reachable from `curTask` in σ_H are subcomponents of the (new) component 21 in memory state σ_G. Similarly, executing `unseal(curTask)` in σ_G result in σ_H.*

2.3 Procedure Calls

Procedure calls are executed by sealing the subheap consisting of objects reachable from any actual parameter and making the new subcomponent the current-component of the callee. Thus, a callee is passed only the sub-component-tree reachable from the actual parameters. Not representing unreachable objects reduces the size of the callee's heap. This increases the reuse of procedure summaries *and* makes the callee analysis more efficient. In this aspect, the $\mathcal{HH}$ semantics is similar to other procedure local-heap semantics [6, 30, 31].

$\mathcal{HH}$, however, allows to further reduce the callee's heap by representing only a subset of the reachable heap. A programmer can specify that certain heap-components are irrelevant for the execution of a procedure. The semantics utilizes this specification by making the irrelevant components opaque: the callee is invoked on a state in which any specified-as-irrelevant component c_{ir} has neither an associated microheap nor subcomponents. When the invoked procedure returns, c_{ir} is reconnected with its microheap and component-subtree, taken as-is from the memory state at the call-site.

The semantics uses the component identifer to determine how to reconnect component c_{ir} to its microheap and component-sub-tree. The callee cannot access, and in particular, mutate the content of opaque components. Thus, the part of the heap represented by c_{ir} and its subtree is not modified by the call. If opaque components would have had the same identifier throughout the execution of a procedure, such a match would be easy. However, it would reduce the possibility for reuse of procedure summaries as it would fix the identifiers of opaque components.

Our semantics allows renaming the identifiers of opaque components. When a procedure is invoked, the semantics maps the identifers of opaque components at the call-site and at the entry-site. The δ function is used to track changes to the identifiers of opaque components *during* the execution of the procedure. On procedure return, the two maps are used to match the identifers at the exit site and at the call site.

Example 5 *Executing `curTask=work.deq()` at state σ_A, specifying that the queue component-elements are irrelevant, results in state σ_D. State σ_E depicts the exit state of `deq`, and σ_G the return state. When the procedure returns, δ and the opaque components' identifiers are used to link the task components turned opaque, back to their microheaps and sub-component-trees. Invoking `curST=this.sbs.deq()` at σ_S, which occurs during the execution of `process`, can reuse the summary computed for the `curTask=work.deq()` invocation by: (i) removing the unreachable task object from the state; (ii) removing the microheap contents of the the subtask-turned-opaque components; and (iii) renaming their identifiers to match those at σ_D.*

We explicitly define δ to show what information needs to be abstracted. In particular, explicitly representing the δ map clarifies the need to abstract the relation of opaque identifers at the procedure's exit-site with their identifiers at the entry-site.

The turn-opaque specification asserts that certain components are irrelevant in a given code block. Here, we only turn objects opaque on procedure calls, but in general, this can be done for arbitrary code blocks.

For simplicity, we require that when a procedure is invoked, it should be *cutpoint-free* [31], i.e., its parameters should dominate the subheap passed to the callee. Thus, *only* the objects pointed to by formal parameters become multiple-headers (*in-ports*) of the callee's current-component, recorded by its microheap's ip partial function. (Graphically, a formal parameter pointing to a node indicates that the node is an in-port.)

Remark. We note that ip can be used to record (unbounded number of) additional entry-points into the callee's heap besides the objects passed as parameter (i.e., cutpoints [30]). However analyzing such procedure calls requires to abstract the sharing patterns between the caller heap and the callee heap, which is not the focus of this paper. We also note that we can assign a canonic identifier to an opaque component using the set of access paths reaching it when the procedure is invoked (i.e., its cutpoint-label [30].) This allows for a storeless [19] representation of $\mathcal{HH}$'s memory states.

2.3.1 Generics and Opaque Heap Components

To benefit from the semantics (and the analysis) we require the programmer to provide a specification for when to `seal` and when to `unseal` components.

When the program utilizes generic implementation of data structures we provide a simplistic approach to automatically generate the specification. Whenever a generic method which is parameterized with a type `T` is invoked, we seal all the parameters that are of type `T` and make all components in the subheap reachable for the procedure whose header is of type `T`, opaque. If the generic procedure returns a value of the `T` type, we unseal it. The specification for our running example is obtained in this way.

2.3.2 Properties of the Semantics

Our generic-specification, as well as the one given by the user, may fail in the sense that a program that can execute in the standard semantics will result in a sealing error in our semantics (or a detection that it is not cutpoint-free).

However, when an execution of our program succeeds, then it results in an equivalent memory state to the one resulting from the standard semantics in the following sense: if all the sealed components are unsealed, the two are isomorphic. Furthermore, our analysis can detect when a sealing error may occur.

3 Abstraction

In this section, we define a parametric abstraction of hierarchical memory states. For brevity, we only discuss the key aspects of the abstraction, formally defined in Appendix C.

We abstract sets of concrete memory states by a point-wise application of an *extraction function* $\beta\colon \Sigma \to \Sigma^\sharp$ (e.g., see [25]) mapping a concrete hierarchical memory state σ to its *best* representation by an *abstract hierarchical memory state* $\sigma^\sharp$. We use set-union as the join-operator.

An abstract state $\sigma^\sharp$ provides a conservative bounded representation for the four unbounded parts of σ: (i) the locations of the microheaps (ii) the nodes of the component-tree (components); (iii) the δ-function tracking the (renaming of) opaque components; and (iv) the labels of edges in the component-tree.

3.1 Abstracting Microheaps

The abstraction of microheaps is similar in flavor to the canonical abstraction of [32] and is determined by a finite set $Loc^\star_{prop}$ of *location properties*, each of which may be thought of as a predicate over a location. Following [28], we define an *abstract location* $l^\sharp$ to be a function from $Loc^\star_{prop}$ to $\{true, false\}$. Given a *microheap*, we represent every concrete location l to an abstract location determined by the set of location properties l satisfies. This guarantees a bounded representation of the microheap. We use a (mandatory) sm property to record the case in which $l^\sharp$ represents more than 1 concrete location, and refer to such an $l^\sharp$ as a *summary* location.

Mapping concrete locations to abstract locations induces a natural abstraction of the intra-microheap link-structure where an edge $\langle l^\sharp_s, f, l^\sharp_t\rangle$ means that the field f of a location l_s represented by $l^\sharp_s$ *may* point to a location l_t represented by $l^\sharp_t$ (when both $l^\sharp_s$ and $l^\sharp_t$ are *not* summary locations, such edges represent a must points-to information).

One of the characteristics of the abstractions used in shape analysis [32] is the use of *non-local properties*, e.g., tracking reachability from variables. A distinguishing feature of the abstraction described here is that such non-local properties are captured in a more controlled fashion, with some consequent advantages, as explained below.

Proving the invariants of some programs may require some information on the contents of subcomponents. Thus, it is sometimes desirable for the abstraction of a microheap (associated with a component) to depend on properties of the subcomponents of the component. An analysis designer can specify a finite set $Digest^{\star}_{prop}$ of *microheap-internal* properties. We define the *digest* of a microheap to be the function from $Digest^{\star}_{prop}$ to $\{true, false\}$ that indicates which of these properties the microheap satisfies. When we abstract a component c, we first annotate its child-ports with the digest and the type of the header of the subcomponent they are linked to in the component-tree. The digest becomes part of the child-port's abstraction. This allows to record in the abstraction of components information about immediate subcomponents. Repeating the above process, allows to record information about deeper component-subtrees, trading microheap reuse for precision.

For example, we can use the state of a subtask as its digest.[2] Thus, in memory state σ_G, the child-port linked to the finished subtask (251) will be abstracted differently than the other child-ports (231 and 241). Location properties can also be dependent on the subcomponent digest. Thus, we can distinguish between queue-nodes holding references to child-ports with different digests. This allows us to infer that the unfinished subtasks precedes the finished subtasks. For more information, see Appendix C.2.

3.2 Abstracting Transparent Components

The abstraction of a transparent component is based on the abstraction of its microheap, augmented with (optional) *macroheap-global* context information. This allows, when desired, to distinguish components based on the *macroheap-global* context in which they appear. (Recall that the abstraction of a microheap is completely independent of the context in which it occurs—except for the controlled dependence on digests associated with child-ports). In particular, we can distinguish between components held in disjoint data structures by tracking from which variables they are reachable. In our running example, the digest allows to distinguish between finished tasks and unfinished tasks, this may not be the case for deeper component-trees.

We allow our abstraction to distinguish between components based on limited context information. Technically, we use a finite set $\mathcal{C}^{\star}_{prop}$ of properties, referred to as *context properties*, in defining the abstraction of components. Context properties are used to capture useful contextual information, such as reachability of components from program variables. We define the *context* of a component c to be a function from $\mathcal{C}^{\star}_{prop}$ to $\{true, false\}$ that indicates which of the context properties are satisfied by c.

An abstract transparent component is a triplet consisting of (i) the context of the component; (ii) the digest of the microheap associated with the component; and (iii) the abstract microheap associated with the component. Note that the context properties are

[2] The digest may be more than the value of a boolean field. For example, a digest of a task can record whether all its subtasks are finished.

associated with an abstract component and *not* with its abstract microheap. This allows sharing the same abstract-microheap in the representation of different components

3.3 Abstracting Opaque Components and the δ-Function

An opaque component does not have a microheap describing its contents. Consequently, the only properties an opaque component has in our semantics are its context properties. Losing the digest information of opaque components may lead to a change in the abstraction of the transparent components, and to a possible loss of precision. E.g., the distinctions made between the finished and unfinished subtasks in the abstraction of σ_S, when `curST=this.sbs.deq()` is invoked, will be lost in the abstraction of memory-state σ_E at `deq`'s entry-site, thus, changing the digest-dependent properties of the queue nodes, and their abstraction.

To prevent this loss of information, we instrument the $\mathcal{HH}$ semantics to record for every opaque component the digest of its microheap at the call site, if the component was transparent at the call-site, or the digest which was recorded for it, if the component was already opaque at the call-site.

We abstract an opaque component using a triplet of functions $\langle c^{ctx}_{prop}, \overline{\delta_{digest}}, \overline{\delta_{ctx}} \rangle$ consisting of: c^{ctx}_{prop}, the (current) context of the component; $\overline{\delta_{digest}}$, the recorded digest for that component, which we refer to as its frozen digest; and $\overline{\delta_{ctx}}$ the context that the component had when the procedure was invoked, which we refer to as the frozen context. Note that a *relational abstraction* [17] *of the δ function* is recorded by the relation between the opaque component's current and frozen properties.

3.4 Abstracting the Component Tree

We abstract the component-tree by merging all components that have the same *abstraction*, thus collapsing the (unbounded) component-tree into a (bounded) *abstract* component-graph. Recall that the edges of the component-tree are labeled by child-ports. Thus, an l^C-labeled edge between a component c_1 and a component c_2 in the component-tree, will be labeled by the abstract location representing l^C in the abstract microheap pertaining to c_1.

The above scheme provides a bounded (parameterized) abstraction for hierarchial memory states. The separation between location (intra-microheap) properties and context-properties allows to reuse the same abstract microheap in different heap-contexts.

Example 6 *Fig. 6 depicts the abstract states pertaining to the concrete states of Fig. 5, when using the properties shown in Tab. 1. (Note that this example does not use digest-properties. See App. C.2 for an example in which these properties are put to use.) The graphical notation is like the one used for depicting concrete memory states in Fig. 6, with the following additions: • Double framed nodes indicates summary locations. All other location-properties that a location satisfies are shown inside its node. • A dotted edge between abstract locations inside a microheap indicates a may point-to information. A solid edge indicates a must point-to information (which is recorded by edges connecting two non-summary nodes). • The context properties and the frozen context properties that a component satisfies are shown inside the component node. Frozen*

$Loc^{all\star}_{prop}$	**Intended Meaning**
$T(l)$	Is the object at location l of type T?
$r_i(l)$	Is location l reachable from in-port number i?
$ils_d(l)$	Is location l pointed-to by a d-field of more than 1 object inside the *microheap*?
$c(l)$	Does l reside on a directed cycle of fields?
$b(l)$	Is the value of the boolean field b of the object at location l true?
$Q(l)$	Do the hd- and tl- fields of object at l point-to the head and the tail of a linked list, respectively?

$Loc^{cur\star}_{prop}$	**Intended Meaning**
$x(l)$	Does the (current) variable `x` point-to location l?
$r_x(l)$	Is location l reachable from the (current) variable `x`?

$\mathcal{C}^{\star}_{prop}$	**Intended Meaning**
$r_x(c)$	Is there a child-port in the current component which is reachable from `x` and from which c can be reached along the hierarchical backbone?

Table 1: Abstraction parameters used in the running example. $Digest^{\star}_{prop} = \emptyset$. The properties in $Loc^{all\star}_{prop}$ are tracked for all locations in all microheaps. The properties in $Loc^{cur\star}_{prop}$ are tracked only for locations in the current microheap. $Loc^{\star}_{prop} = Loc^{all\star}_{prop} \cup Loc^{cur\star}_{prop}$.

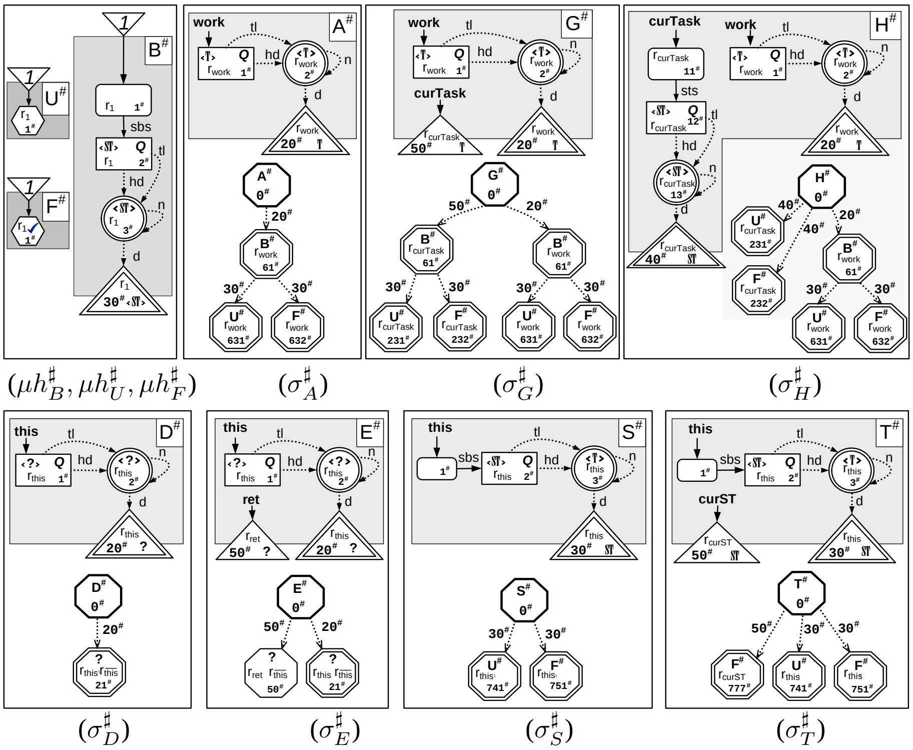
context properties are drawn with an overline. The double frame around all the components, except the current component nodes, and the dotted tree edges emphasize that our abstraction does not record summary information for components. (We conservatively assume that all abstract components, but the current component, may represent multiple concrete components.)

The labels on the edges of the component-graph are the identifiers of the abstract child-ports inside the abstract microheap linked to the abstract component. Identifiers have no meaning in the abstraction. They are merely used for notational convenience. However, they indicate the possibility of a shared representation of abstract microheaps in and across abstract memory states, as described below.

Note that the abstract microheaps $\mu h^{\sharp}_{B}$, $\mu h^{\sharp}_{U}$, and $\mu h^{\sharp}_{F}$ are used in the representation of 5 of the abstract memory states. Furthermore, mutations done by `iterate` *and* `process` *neither effect these abstract microheaps nor require to consider their internal structure (e.g., for computing reachability from variables). Also note that because* `deq` *is summarized using opaque components, the same abstract summary $\sigma^{\sharp}_{D} \mapsto \sigma^{\sharp}_{E}$ can be reused to compute the effect of invoking* `deq` *on the abstract memory states $\sigma^{\sharp}_{A}$ and $\sigma^{\sharp}_{S}$.*

3.5 Unsealing Components

When a component is `unsealed`, the analysis absorbs a copy of the abstract microheap associated with the unsealed component into the abstract microheap of the current component. For example, when the component pointed to by `curTask` in $\sigma^{\sharp}_{G}$ is unsealed it



$(\mu h^\sharp_B, \mu h^\sharp_U, \mu h^\sharp_F)$ $(\sigma^\sharp_A)$ $(\sigma^\sharp_G)$ $(\sigma^\sharp_H)$

$(\sigma^\sharp_D)$ $(\sigma^\sharp_E)$ $(\sigma^\sharp_S)$ $(\sigma^\sharp_T)$

Figure 6: Abstract memory states occurring in the running example.

results in $\sigma^\sharp_H$. Note that the properties of the locations in the absorbed microheap can be determined from their location-properties within the microheap, and the properties of the child-port pointed to by `curTask` at the current-component. The fact that the subcomponent of the absorbed task are disjoint from those of the tasks held in the `work` queue is guaranteed by the macroheap tree decomposition.

4 Related Work

Existing storeless-semantics represent either all the dynamically allocated objects [5, 19] or all the objects in the procedure's local-heap [30]. The $\mathcal{HH}$ semantics is novel in the sense that it represents only a subset of the procedure's local-heap. Existing storeless semantics provide a flat graph model of the heap. In contrast, $\mathcal{HH}$ provides an hierarchical graph model.

The interprocedural shape analysis algorithms in [6, 10, 12, 18, 30] aim for maximal precision by tabulating correlated abstractions of all the objects in the procedure's local-heap. [14] aims for maximal reuse of procedure's summaries by tabulating abstractions containing a single (abstract) object. Our work can be thought of as an attempt to strike a balance between these two extremes by tabulating correlated abstractions of a *subset* of the objects in the procedure's local-heap. We note that aforementioned approaches do not require a user specification, which, in general, is required in our work.

[20, 33] utilize specified pre- and post- conditions to achieve modular shape analysis. Our analysis requires a specification for the irrelevant parts. We provide a simplistic approach to generate the specification for programs manipulating generic implementations of data structures. We hope to harness a preliminary analysis to automatically determine the specification in other cases, as done, e.g., in [14].

Existing shape abstractions, e.g., [28, 32], do not explicitly limit the effect mutations have on the abstraction, with the list-abstraction of [11, 23] being a notable exception. In contrast, our abstraction allows to control the degree in which mutations in one component (containing an arbitrary data structure) effect the abstractions of other components.

[21] presents a heap abstraction for hierarchical data structures based on grammars. Our component-tree can be used to explicitly represent the data structure hierarchy. It seems natural to augment its abstraction with (digest-based) grammar-properties.

In local reasoning [16, 29], assertions regarding disjoint portions of a heap can be combined into an assertion about the whole heap using a common variable environment. In our work, abstract microheaps correspond to local-assertions and the abstract macroheap to an (abstraction) of the common environment.

Our tree-decomposition of the heap is inspired by the works concerning *encapsulation* (also known as *confinement* or *ownership*) [1–4, 7, 8, 13, 15, 22, 24, 26, 27]. We believe that similar type annotations can be used as a specification for our analysis.

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*, 1997.

[2] A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *Symp. on Princ. of Prog. Lang.*, 2002.

[3] B. Bokowski and J. Vitek. Confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1999.

[4] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-03)*, volume 38:1 of *ACM SIGPLAN Notices*, pages 213–223, New York, January 15–17 2003. ACM Press.

[5] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 55–65. ACM Press, 2003.

[6] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.

[7] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming*, 2001.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1998.

[9] D. Clément, J. Despeyroux, Th. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. In *LISP and Functional Programming*, pages 13–27, 1986.

[10] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[11] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[12] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

[13] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2001.

[14] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symp. on Princ. of Prog. Lang.*, 2005.

[15] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Phoenix, Arizona, 1991.

[16] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[17] B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *SAS*, 2005.

[18] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium*, 2004.

[19] H.B.M. Jonkers. Abstract storage structures. In de Bakker and van Vllet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[20] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *In 14th International Conference on Compiler Construction (tool demo)*, 2005.

[21] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

[22] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2002.

[23] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, 2005.

[24] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[25] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[26] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *The First International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2003. available at http://www.cs.uu.nl/∼dave/iwaco/index.html.

[27] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[28] Andreas Podelski and Thomas Wies. Boolean heaps. In *SAS*, 2005.

[29] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, 2002.

[30] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Princ. of Prog. Lang.*, 2005.

[31] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS)*, 2005.

[32] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[33] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *VMCAI 2006*, 2006.

```
public class Main {
public static void main(String argv[]) {
    GQueue⟨Task⟩ pending =
                new GQueue⟨Task⟩();
    GQueue⟨Task⟩ completed =
                new GQueue⟨Task⟩();
    genTasks(pending);

    while (!pending.isEmpty())
        Task did = iterate(pending)
        completed.enq(did);
    }
}

private static Task iterate(
            GQueue⟨Task⟩ work) {
    while (true) {
        Task curTask = work.deq();
        // unseal(curTask);
        if(curTask.process())
            return curTask;
        else
            // seal(curTask);
            work.enq(curTask);
    }
}

private static void genTasks(
        GQueue⟨Task⟩ pending) {
    for (int i=0; i < 3; i++) {
        Task task = new Task();
        for (int j=0; j < 3; j++) {
            SubTask st = new SubTask();
            task.add(st);
        }
        // pack(task);
        pending.enq(task);
    }
}
}
```

```
public class Task {
private GQueue⟨SubTask⟩ sts;

public boolean process() {
    boolean taskDone = false;
    while (... && !taskDone) {
        SubTask curST = this.sbs.deq();
        // unseal(curST);
        if (curST.isDone())
            taskDone = true;
        else
            curST.execute();
        // seal(curST);
        sbs.enq(curST);
    }
    return taskDone;
}

public Task() {
    subtasks = new GQueue⟨SubTask⟩();
}
}

public class SubTask {
private boolean isDone = false;
public void execute() {
    ...
    this.isDone = true;
}
}
```

```
GQueueNode⟨T⟩ head, tail;

public GQueue() {
    this.head = null;
    this.tail = null;
}

public void enq(T o) {
    GQueueNode⟨T⟩ t =
            new GQueueNode⟨T⟩();
    t.d = o;
    if (head == null) {
        t.next = null;
        this.head = t;
        this.tail = t;
    }
    else {
        GQueueNode⟨T⟩ last = this.tail;
        last.next = t;
        this.tail = t;
    }
}

public T deq() {
    if (this.head == null)
        throw new InternalError();

    GQueueNode⟨T⟩ first = this.head;
    this.head = first.next;
    if (this.head == null)
        this.tail = null;
    return first.d;
}

public boolean isEmpty() {
    return this.head == null;
}
}

public class GQueueNode⟨E⟩ {
    GQueueNode⟨E⟩ n;
    E d;
}
```

Figure 7: The running example including code which was omitted in Fig. 1.

A Additional Code

Fig. 7 shows the running example including code which was omitted from Fig. 1. The actual condition for preemption of an uncompleted task (in method `process`) and the code used for the actual execution of subtasks (in method `execute`) are irrelevant, and thus, not shown.

$$
\begin{array}{llll}
P & \in & \mathit{prog} & ::= \ \overline{\mathit{rcdecl}}\ \overline{\mathit{fndecl}} \\
 & & \mathit{rcdecl} & ::= \ \texttt{record}\ t := \{\,\overline{\mathit{tname}\,f}\} \mid \texttt{record}\ t\langle t'\rangle := \{\,\overline{\mathit{tname}\,f}\} \\
 & & \mathit{tname} & ::= \ \texttt{int} \mid t \\
 & & \mathit{prdecl} & ::= \ p(\overline{\mathrm{tname}\ x}) := \overline{\mathit{vdecl}}\ st \\
 & & \mathit{vdecl} & ::= \ \mathit{tname}\ \mathrm{VarId} \\
st & \in & \mathit{stms} & ::= \ x{=}c \mid x{=}y \mid\ x{=}y\ \mathit{op}\ z \mid x{=}y.f \mid \\
 & & & \ \ x.f{=}\ \texttt{null} \mid x.f{=}y \mid x = \texttt{alloc}\ t \mid \\
 & & & \ \ p(\overline{x}) \mid p\langle t\rangle(\overline{x}) \mid \mathit{lb}\colon\ st \mid \texttt{while}\ (\mathit{cnd})\ \texttt{do}\ st\ \texttt{od} \mid \\
 & & & \ \ st\,;st \mid \texttt{if}\ (\mathit{cnd})\ \texttt{then}\ st\ \texttt{else}\ st\ \texttt{fi} \mid \\
 & & & \ \ \texttt{seal}(x) \mid \texttt{unseal}(x) \\
 & & \mathit{cnd} & ::= \ x == y \mid x\,! = y \mid x == c \mid x\,! = c \\
c & \in & \mathit{const} & ::= \ \texttt{null} \mid \texttt{n}
\end{array}
$$

Figure 8: Syntax of **HAlgol**.

A.1 Syntax of HAlgol

In this section, we introduce a simple imperative language called **HAlgol**. Programs in **HAlgol** consist of a collection of procedures including a main function. The programmer can also define her own types (à la C structs) and refer to heap-allocated objects of these types using pointer variables. Parameters are passed by value. Formal parameters cannot be assigned to.

The syntax of **HAlgol** is defined in Fig. 8. The notation $\bar{z}$ denotes a sequence of z's. We define the syntactic domains $x, y \in \mathit{VarId}$, $f \in \mathcal{F}$, $p \in \mathit{ProcId}$, $t \in \mathcal{T}$, and $lb \in \mathit{Labels}$ of variables, field names, procedure identifiers, type names, and program-labels, respectively.

HAlgol provides an explicit handling of components: Executing $\texttt{seal}(x)$ packs all the objects and component instances in the sub-state reachable from x. These objects are removed from the memory state and placed inside a new component. The object which was pointed to by x changes its role and becomes the *child*-port connected to the new component. Applying the $\texttt{unseal}(x)$ commands unseals the component linked to the child-port pointed to by x and integrates its contents with the current component.

Generic data structures can be written in **HAlgol** by defining parametric user-defined types (with a single type parameter, written inside angle brackets). Functions can also be parameterized with a type parameter. The latter is also used as a simple form of a specification: Defining a function as $p\langle T\rangle$ indicates that when $p\langle T\rangle$ is invoked, all reachable headers of the (instantiated) type T (and their subcomponents) are irrelevant to the function. For simplicity, we assume that this is always the case for parametric functions.

$$
\begin{array}{ll}
\langle \mathtt{x} = \mathtt{null}, \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho[x \mapsto \textit{null}], {}_{M}h, \delta \rangle & \\
\langle \mathtt{x} = \mathtt{y}, \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho[x \mapsto \rho(y)], {}_{M}h, \delta \rangle & \\
\langle \mathtt{x} = \mathtt{y.f}, \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho[x \mapsto h(\rho(y), f)], {}_{M}h, \delta \rangle & \rho(y) \in L^O \\
\langle \mathtt{x.f} = \mathtt{null}, \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho, \mathit{updRoot}({}_{M}h, \rho(x), f, \textit{null}), \delta \rangle & \rho(x) \in L^O \\
\langle \mathtt{x.f} = \mathtt{y}, \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho, \mathit{updRoot}({}_{M}h, \rho(x), f, \rho(y)), \delta \rangle & \rho(x) \in L^O \\
\langle \mathtt{x} = \mathtt{alloc}\ \mathtt{T}(), \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho[x \mapsto l], \langle c, C, e, d[c \mapsto \mu h'] \rangle, \delta \rangle & l \in \mathit{Loc} \setminus (L^O \cup L^C) \\
\quad \text{where: } \mu h' = \langle \mathit{ip}, L^O \cup \{l\}, L^C, h, t[l \mapsto T] \rangle & \\
\langle \mathtt{nop}, \langle \rho, {}_{M}h, \delta \rangle \rangle \overset{H}{\rightsquigarrow} \langle \rho', {}_{M}h', \delta' \rangle & \langle \rho, {}_{M}h, \delta \rangle \sim \langle \rho', {}_{M}h', \delta' \rangle
\end{array}
$$

Figure 9: Axioms for intra-procedural statements. We use ${}_{M}h$ as a shorthand for $\langle c, C, e, d \rangle$. We denote $d(c)$ by $\langle \mathit{ip}, L^O, L^C, h, t \rangle$; We assume that variables and fields are nullified (assigned *null*) before being assigned any other value and that the pointer fields of allocated objects are nullified. (Alternatively, the assumed nullifications could have been built-in into the semantics.)

B Large-Step Operational Semantics

Fig. 9 provides the semantics of intraprocedural statements. Fig. 10 provides the semantics of interprocedural statements. Fig. 11 provides the semantics of the `seal` and `unseal` statements. Auxiliary functions are defined in figures 12, 13, 14, and 15.

$$\frac{\langle \text{body of } p\langle ? \rangle, \sigma_e\rangle \overset{H}{\rightsquigarrow} \sigma_x}{\langle p\langle T\rangle(x_1,\ldots,x_k), \sigma_c\rangle \overset{H}{\rightsquigarrow} \sigma_r} \quad \{\rho(x_i) \mid 1 \leq i \leq k\}\ \mathit{dominatingSet}\ (\sigma_c)$$

$$\begin{array}{ll}
\mathit{where} & \\
\quad \mathit{seps} \in \{\rho_c(x_i) \mid 1 \leq i \leq k\}^{\{1,\ldots,|\{\rho_c(x_i)|1 \leq i \leq k\}|\}} & \text{bijective} \\
\quad \langle \mu h_e, \mu h_{ctx}\rangle = \mu\mathit{cut}(\mathit{seps}, d_c(c_c)) & \\
\quad c_{new} \in \mathcal{C} \setminus C_c & \sigma_x \sim \sigma_{\tilde{x}} \\
\quad {}_{M}h_c^{preCall} = \mathit{demote}({}_{M}h_c, \mathit{seps}, c_{new}, \mu h_e, \mu h_{ctx}) & \quad \mathit{locs}(\mu h_{ctx}) \cap \mathit{locs}(d_{\tilde{x}}(c_{\tilde{x}})) = \emptyset, \\
\quad \langle {}_{M}h_{ctx}, \{{}_{M}h_c^{rel}\}\rangle = {}_{M}\mathit{cut}({}_{M}h_c^{preCall}, \{c_e\}) & C_c \cap C_{\tilde{x}} = \emptyset \\
\quad \langle {}_{M}h_e, {}_{M}H_T\rangle = {}_{M}\mathit{cut}({}_{M}h_c^{rel}, C_T) & {}_{M}h_x^{reach} = {}_{M}\mathit{paste}({}_{M}h_{\tilde{x}}, {}_{M}H_T, \mathit{id}[\delta_{\tilde{x}}^{-1}]) \\
\quad \quad C_T = \{c \in \mathcal{C} \mid \mathit{hdType}(d_{\tilde{c}}^{rel}(c)) = T\} & {}_{M}h_r^{withCtx} = {}_{M}\mathit{paste}({}_{M}h_{ctx}, \{{}_{M}h_x^{reach}\}, \mathit{id}[c_e \mapsto c_{\tilde{x}}]) \\
\quad \delta_e = \delta_c|_{C_e}[c \mapsto c \mid c \in C_T] & \mu h_r = \mu\mathit{paste}(\mathit{seps}, \mu h_{ctx}, \mu h_{\tilde{x}}) \\
\quad \sigma_e = \langle [z_i \mapsto \rho_{\tilde{c}}^{rel}(x_i) | 1 \leq i \leq k], {}_{M}h_e, \delta_e\rangle & \sigma_r = \langle \rho_c, \mathit{promote}({}_{M}h_r^{withCtx}, c_r, \mu h_r)\rangle
\end{array}$$

Figure 10: Inference rule for function invocation in the $\mathcal{HH}$ semantics, assuming the formal variables of p are $z_1, \ldots, z_k$ and that p's return value is a pointer. We use σ_c as shorthand for $\langle \rho_c, {}_{M}h_c, \delta_c\rangle$, ${}_{M}h_c$ as a shorthand for $\langle c_c, C_c, e_c, d_c\rangle$, μh_c as a shorthand for $\langle ip_c, {L^O}_c, {L^C}_c, h_c, t_c\rangle$, and L_c as a shorthand for $L_c^O \cup L_c^C$ (and similarly for σ_e, σ_c, and σ_r). We implicitly use the primed resp. subscript of a state to refer to its current microheap, i.e., $\mu h_c = d_c(c_c)$.

$$\begin{array}{ll}
\langle \texttt{seal(x)}, \langle \rho, \langle c, C, e, d\rangle, \delta\rangle\rangle \overset{H}{\rightsquigarrow} \langle \rho, \langle c, C', e', d'\rangle, \delta\rangle & \{\rho(x)\}\ \mathit{dominatingSet}\ (\langle \rho, \langle c, C, e, d\rangle\rangle) \\
\quad \text{where:} & \\
\quad \quad \langle \mu h_{rt}, \mu h_{ct}\rangle = \mu\mathit{cut}(\lambda 1.\rho(x), d(c)) & \\
\quad \quad c_{ct} \in \mathcal{C} \setminus C & \\
\quad \quad {}_{M}h' = \mathit{demote}({}_{M}h, \lambda 1.\rho(x), c_{ct}, \mu h_{ct}, \mu h_{rt}) & \\
\langle \texttt{unseal(x)}, \langle \rho, \langle c, C, e, d\rangle, \delta\rangle\rangle \overset{H}{\rightsquigarrow} \langle \tilde{\rho}, \mathit{promote}(\langle \tilde{c}, \tilde{C}, \tilde{e}, \tilde{d}\rangle, \tilde{c}_{unsealed}, \mu h_{\text{newRoot}}), \delta\rangle & \rho(c) \in L^C \\
\quad \text{where:} & \\
\quad \quad \sigma \sim \langle \tilde{\rho}\langle \tilde{c}, \tilde{C}, \tilde{e}, \tilde{d}\rangle\rangle,\ \tilde{c}_{unsealed} = \tilde{e}(\tilde{c})(\tilde{\rho}(x)),\ \mathit{locs}(\tilde{d}(\tilde{c})) \cap \mathit{locs}(\tilde{d}(\tilde{c}_{unsealed})) = \emptyset & \\
\quad \quad \mu h_{\text{newRoot}} = \mu\mathit{paste}(\lambda 1.\tilde{\rho}(x), \tilde{d}(\tilde{c}), \tilde{d}(\tilde{c}_{unsealed})) &
\end{array}$$

Figure 11: Axioms for seal and unseal statements.

$$
\begin{array}{l}
\mu cut : \mathcal{IP} \times \mu\mathcal{H} \rightharpoonup \mu\mathcal{H} \times \mu\mathcal{H}\ \ s.t. \\
\quad \mu cut(ip_{packed}, \langle ip, L^O, L^C, h, t\rangle) = \langle \mu h_{in}, \mu h_{out}\rangle \\
\qquad \text{where:} \\
\qquad\quad L_{seps} = range(ip_{packed}) \\
\qquad\quad L_{packed} = \mu reach(L_{seps}, h) \\
\qquad\quad \mu h_{in} = \langle ip_{packed}, L_{packed} \cap L^O, L_{packed} \cap L^C, h|_{L_{packed}}, t|_{L_{packed}}\rangle \\
\qquad\quad \mu h_{out} = \langle ip, L^O \setminus L_{packed}, L^C \setminus L_{packed} \cup L_{seps}, h|_{L^O \setminus L_{packed}}, t|_{L^O \setminus L_{packed}}\rangle \\
\mu paste : \mathcal{IP} \times \mu\mathcal{H} \times \mu\mathcal{H} \rightharpoonup \mu\mathcal{H}\ \ s.t. \\
\quad \mu paste(ip_{rc}, \mu h_{\text{root}}, \mu h_{\text{child}}) = \langle ip_{\text{root}}, L^O_{\text{newRoot}}, L^C_{\text{newRoot}}, h_{\text{newRoot}}, t_{\text{newRoot}}\rangle \\
\qquad \text{where:} \\
\qquad\quad L_{rootSeps} = range(ip_{packed}) \\
\qquad\quad L_{childInports} = range(ip_{child}) \\
\qquad\quad L^O_{newRoot} = L^O_{root} \cup \{ip_{root}(i) \mid ip_{child}(i) \in L^O_{child}\} \cup L^O_{child} \setminus L_{childInports} \\
\qquad\quad L^C_{newRoot} = L^C_{root} \cup \{ip_{root}(i) \mid ip_{child}(i) \in L^C_{child}\} \cup L^C_{child} \setminus L_{childInports} \\
\qquad\quad h_{newRoot} = h_{root} \cup h_{child}|_{L^O_{child} \setminus L_{childInports}} \cup \lambda l \in L_{rootSeps}.h_{child}(ip_{child}(ip^{-1}_{rc}(l))) \\
\qquad\quad t_{newRoot} = t_{root} \cup t_{child}|_{L^O_{child} \setminus L_{childInports}} \cup \lambda l \in L_{rootSeps}.t_{child}(ip_{child}(ip^{-1}_{rc}(l)))
\end{array}
$$

Figure 12: Functions for disassembly (μcut) and assembly ($\mu paste$) of micro-heaps.

$$
\begin{array}{l}
{}_{M}cut \in {}_{M}\mathcal{H} \times \mathcal{C} \rightharpoonup ({}_{M}\mathcal{H} \times 2^{M\mathcal{H}})\ s.t. \\
\quad {}_{M}cut(\langle c, C, e, d\rangle, C_c) = \langle\langle c, C_{rt} \cup C_c, e|_{C_{rt}}, d|_{C_{rt}}\rangle, \{{}_{M}project(\langle c, C, e, d\rangle, c_c) \mid c_c \in C_c\}\rangle \\
\qquad where \\
\qquad\quad C_{rt} = C \setminus {}_{M}reach(C_c, e) \\
{}_{M}paste \in {}_{M}\mathcal{H} \times 2^{M\mathcal{H}} \rightharpoonup {}_{M}\mathcal{H}\ s.t. \\
\quad {}_{M}paste(\langle c_{rt}, C_{rt}, e_{rt}, d_{rt}\rangle, {}_{M}H, \delta) = \langle c_{rt}, C', e', d'\rangle \\
\qquad C' = dom(d_{rt}) \cup \bigcup_{\langle c,C,e,d\rangle \in MH} C \\
\qquad \delta \circ e_{rt} \cup \bigcup\{e \mid \langle c, C, e, d\rangle \in {}_{M}H]\} \\
\qquad d_{rt} \cup \bigcup\{d \mid \langle c, C, e, d\rangle \in {}_{M}H\}
\end{array}
$$

Figure 13: Functions for disassembly (${}_{M}cut$) and assembly (${}_{M}paste$) of hierarchical-heaps.

$$
\begin{array}{l}
demote \in {}_{M}\mathcal{H} \times \mathcal{IP} \times \mathcal{C} \times \mu\mathcal{H} \times \mu\mathcal{H} \rightharpoonup {}_{M}\mathcal{H}\ s.t. \\
\quad demote(\langle c, C, e, d\rangle, ip, c_{rel}, \mu h_{rel}, \mu h_{ctx}) = \langle c, C \cup \{c_{rel}\}, e_{rel}, d[c_{rel} \mapsto \mu h_{rel},\ c \mapsto \mu h_{ctx}]\rangle \\
\qquad where \\
\qquad\quad e_{rel} = e[c_{rel} \mapsto e(c)|_{locs(\mu h_{rel})}[l \mapsto c_{rel} \mid l \in range(ip)],\ c \mapsto e(c)|_{locs(\mu h_{ctx})}] \\
promote \in {}_{M}\mathcal{H} \times \mathcal{C} \times \mu\mathcal{H} \rightharpoonup {}_{M}\mathcal{H}\ s.t. \\
\quad promote(\langle c, C, e, d\rangle, c_{cd}, \mu h_{rt}) = \langle c, C \setminus \{c_{cd}\}, e', d|_{C\setminus\{c_{cd}\}}[c \mapsto \mu h_{rt}]\rangle \\
\qquad where \\
\qquad\quad e' = e|_{C\setminus\{c_{cd}\}}[c \mapsto (e(c) \cup e(c_{cd}))]
\end{array}
$$

Figure 14: Demotion and promotion of sub states.

$\mathit{updRoot} \in {}_{M}\mathcal{H} \times \mathit{Loc} \times \mathcal{F} \times \mathit{Val} = {}_{M}\mathcal{H}\ \mathit{s.t.}$

$\quad \mathit{updRoot}(\langle c, C, e, d\rangle, l, f, v) = \langle c, C, e, d[c \mapsto \langle \mathit{ip}, L^{O}, L^{C}, h[l \mapsto h(l)[f \mapsto v], t\rangle]\rangle$

$\qquad \mathit{where}$

$\qquad\quad \langle \mathit{ip}, L^{O}, L^{C}, h, t\rangle = d(c)$

$\mathit{locs} : \mu\mathcal{H} \to 2^{\mathit{Loc}}\ \mathit{s.t.}$

$\quad \mathit{locs}(\langle \mathit{ip}, L^{O}, L^{C}, h, t\rangle) = L^{O} \cup L^{C}$

$\mathit{hdType} : \mu\mathcal{H} \to \mathcal{T}\ \mathit{s.t.}$

$\quad \mathit{hdType}(\langle \mathit{ip}, L^{O}, L^{C}, h, t\rangle) = t(\mathit{ip}(1))$

$\mu\mathit{reach} : 2^{\mathit{Loc}} \times \mathcal{H} \to 2^{\mathit{LOC}}\ \ \mathit{s.t.}$

$\quad \mu\mathit{reach}(L, h) = \mathit{lfp}(\lambda X. L \cup X \cup \{h(l)(f) \in \mathit{Loc} \mid l \in X\})$

${}_{M}\mathit{reach} : 2^{\mathcal{C}} \times \mathcal{E} \to 2^{\mathcal{C}}\ \ \mathit{s.t.}$

$\quad {}_{M}\mathit{reach}(C, e) = \mathit{lfp}(\lambda X. C \cup X \cup \{e(c)(l) \mid c \in X,\ l \in \mathit{Loc}\})$

$\mathit{dominatingSet} \subseteq 2^{\mathit{Loc}} \times \Sigma\ \ \mathit{s.t.}$

$\quad L\ \mathit{dominatingSet}\ (\langle \rho, \langle c, C, e, d\rangle\rangle) \iff (L_{\mathit{roots}} \cup L_{\mathit{pcp}}) \cap (L_{\mathit{reach}} \setminus L) = \emptyset$

$\qquad \mathit{where}$

$\qquad\quad \langle \mathit{ip}, L^{O}, L^{C}, h, t\rangle = d(c)$

$\qquad\quad L_{\mathit{roots}} = \mathit{range}(\rho) \cup \mathit{range}(\mathit{ip})$

$\qquad\quad L_{\mathit{reach}} = \mu\mathit{reach}(L, h)$

$\qquad\quad L_{\mathit{pcp}} = \{h(l)(f) \in \mathit{Loc} \mid l \notin L_{\mathit{reach}},\ f \in \mathcal{F}\}$

${}_{M}\mathit{project} \in {}_{M}\mathcal{H} \times \mathcal{C} \rightharpoonup {}_{M}\mathcal{H}\ \mathit{s.t.}$

$\quad {}_{M}\mathit{cut}(\langle c, C, e, d\rangle, c') = \langle c', C_{\mathit{rel}}, e|_{C_{\mathit{rel}}}, d|_{C_{\mathit{rel}}}\rangle$

$\qquad \mathit{where}$

$\qquad\quad C_{\mathit{rel}} = {}_{M}\mathit{reach}(\{c'\}, e)$

$\sim\ \subseteq\ \Sigma \times \Sigma\ \ \mathit{s.t.}$

$\quad \langle \rho_1, \langle c_1, C_1, e_1, d_1\rangle, \delta_1\rangle \sim \langle \rho_2, \langle c_2, C_2, e_2, d_2\rangle, \delta_2\rangle \iff$

There exists a bijective function $m : C_1 \to C_2$ s.t.

$\quad m(c_1) = (c_2)$

$\quad m|_{\mathit{dom}(d_1)} \in \mathit{dom}(d_1) \to \mathit{dom}(d_2)$

$\quad \forall c \in \mathit{dom}(d_1)$: there exists a function $m_c : \mathit{locs}(d_1(c)) \to \mathit{locs}(d_2(m(c)))\ \ \mathit{s.t.}$

$\qquad m_c|_{\mathit{Val} \setminus \mathit{Loc}} = \lambda v.v$

$\qquad d_1(c) \sim_{m_c} d_2(m(c))$

$\qquad \forall c, c' \in C_1, \forall l \in \mathit{Loc} : c' = e_1(c)(l) \iff m(c') = e_2(m(c))(m_c(l))$

$\qquad m_{c_1}(\rho_1(x)) = \rho_2(x)$ for all $x \in \mathit{VarId}$

$\sim_{.} \subseteq\ (\mathit{Val} \rightharpoonup \mathit{Val}) \times (\mu\mathcal{H} \times \mu\mathcal{H})\ \ \mathit{s.t.}$

$\quad \langle \mathit{ip}_1, {L^{O}}_1, {L^{C}}_1, h_1, t_1\rangle \sim_m \langle \mathit{ip}_2, {L^{O}}_2, {L^{C}}_2, h_2, t_2\rangle \iff$

$\quad m|_{L_1^{O}} \in L_1^{O} \to L_2^{O}$ is a bijective function

$\quad m|_{L_1^{C}} \in L_1^{C} \to L_2^{C}$ is a bijective function

$\quad m|_{\mathit{Val} \setminus \mathit{Loc}} = \lambda v.v$

$\quad \mathit{dom}(\mathit{ip}_1) = \mathit{dom}(\mathit{ip}_2)$ and $\forall i \in \mathit{dom}(\mathit{ip}_1) : m(\mathit{ip}_1(i)) = \mathit{ip}_2(i)$

$\quad \forall l \in L_1^{O} : \mathit{dom}(h_1(l)) = \mathit{dom}(h_2(m(l)))$ and

$\qquad\qquad \forall f \in \mathit{dom}(h_1(l)) : m(h_1(l)(f)) = h_1(m(l))(f)$

Figure 15: Auxiliary functions.

C The Hierarchial Heaps Abstract Domain

Appendix C.1 elaborates on the intricacies regarding the δ-function, its abstraction, and the abstraction of opaque components. Appendix C.2 gives an example for an abstraction which uses digests. Appendix C.3 formally defines the abstract domain and the abstraction function.

C.1 The δ-function and the Abstraction of Opaque Components

C.1.1 The δ-function

The effect of a procedure on the contents and the component-subtrees of transparent components could be deduced from the memory state at the exit-site of the callee. However, the only effect a procedure has on opaque components is on the inter-component references to them from transparent components. In order for the caller to re-associate opaque components with their cutoff content and component-subtree, we need to correlate the opaque components (in the possibly mutated macroheap) of the invoked procedure exit-state to the opaque components at its entry-state, and thus, to the call-state. For this purpose, we explicitly introduced a designated function δ (see Sec. 2.3). The function δ is a bijection. It maps an opaque component at procedure exit to an opaque component from the entry of the procedure.

C.1.2 Abstracting Opaque Components using an Instrumented Concrete Semantics

An opaque component does not have a microheap describing its contents. Consequently, the abstraction of an opaque component consists only of its context, frozen digest (as propagated from the caller that made this component opaque), and frozen context (see Sec. 3.) The information required to deduce the frozen properties is not maintained in the memory state, as defined in Sec. 2.1. Thus, we add the required information by instrumenting the concrete semantics.

We instrument the concrete semantics with the frozen context- and digest- properties of opaque components. Specifically, we augment the concrete memory states with a pair of (immutable) functions $\langle \overline{\delta_{digest}}, \overline{\delta_{ctx}} \rangle \in \overline{\Delta} = \langle \mathcal{C} \rightharpoonup \mu\mathcal{H}^{Digest}_{prop}, \mathcal{C} \rightharpoonup \mathcal{C}^{ctx}_{prop} \rangle$ mapping opaque components to frozen properties. The partial function $\overline{\delta_{digest}}$ maps an opaque component to its frozen digest. The partial function $\overline{\delta_{digest}}$ maps an opaque component to its frozen context. Both functions are defined for, and only for, the opaque components in the memory state.

For example, a $\overline{\delta_{ctx}}$ based on the context-properties shown in Tab. 1 will record that all the opaque components in memory state σ_D, shown in Fig. 5, were reachable from the `this` variable when `deq` was invoked.

A $\overline{\delta_{digest}}$ function recording the single digest property of Fig. 16(a) maps opaque component 41 in σ_D (when resulting from invoking `deq` in memory state σ_S) to $true$, and the other opaque components, i.e., 21 and 31, to $false$.

C.2 Abstraction with Digest Information

The digest, as described in Sec. 3, is computed only based on internal properties of the microheaps. This allows to record in the abstraction of components information about immediate subcomponents. However, the process of computing a digest, and annotating child-ports with them can be repeated several time.

Every repetition allows to record information about subcomponents one level deeper in the components-tree. Additional properties at the digest, trades space complexity for added precision. Fixing the number of properties in a digest, trades a more elaborate information regarding shallower components for a deeper horizon.

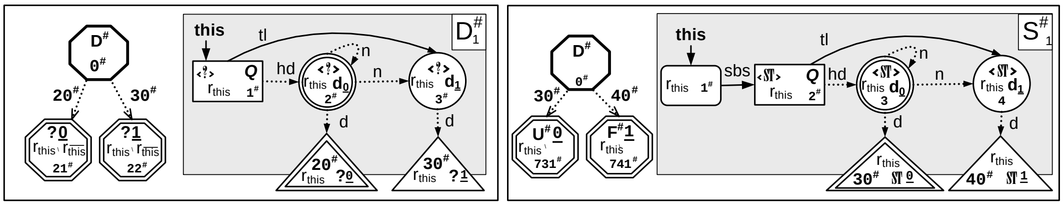
Example 7 *The abstract states $\sigma^{\sharp}_{D_1}$ and $\sigma^{\sharp}_{S_1}$ shown in Fig. 16 represent the concrete states σ_D and σ_S of Fig. 5, when using the properties of Fig. 16(a).*

We label components and child-ports with the 1-bit property vector pertaining to the (single) $\mathit{finishedST}$ digest-property (i.e., either $\overline{0}$ or $\overline{1}$). We indicate whether an abstract queue node has the digest-dependent queue-node property $\mathit{digestAt}_d(l) = \underline{b}$ by labeling the queue node with $e_{\underline{b}}$.

Note that the digest allows us to infer that in all concrete memory states represented by abstract state $\sigma^{\sharp}_{S_1}$, all finished subtasks precede the unfinished ones in the queue. Similarly, in the generic queue, we can infer that all the elements with digest $\underline{0}$ precede the one with digest $\underline{1}$. This allows us to verify that the generic `deq` will return an opaque component whose digest is $\underline{0}$, which the caller, the `process` method, knows to be an unfinished suntask.

$\mathit{Digest}^{\star}_{prop}$	**Intended Meaning**
$\mathit{finishedST}$	The microheap's header is a finished `SubTask`
$\mathit{Loc}^{all\star}_{prop}$	**Intended Meaning**
$\mathit{digestAt}_d(l) = \underline{b}$	Does the element pointed to by `d`-field of the queue node in location l point-to a child port linked to a component whose microheap has digest $\underline{b}$?

(a) Additional digest-related abstraction parameters



$(\sigma^{\sharp}_{D_1})$ $(\sigma^{\sharp}_{S_1})$

(b) Abstract states with digest-related properties.

Figure 16: Abstract memory states with additional properties pertaining to digests.

Parameters to the abstraction	**Description**
$\mathit{digest} \in \mu\mathcal{H}^{\mathit{Digest}}_{\mathit{prop}} = \{\mathit{false}, \mathit{true}\}^{\lvert \mathit{Digest}^{\star}_{\mathit{prop}} \rvert}$	Externally visible properties of microheaps
$l_{\mathit{prop}} \in \mathit{Loc}_{\mathit{prop}} = \{\mathit{false}, \mathit{true}\}^{\lvert \mathit{Loc}^{\mathit{cur}\star}_{\mathit{prop}} \cup \mathit{Loc}^{\mathit{all}\star}_{\mathit{prop}} \rvert}$	Tracked properties of locations
$c^{\mathit{ctx}}_{\mathit{prop}} \in \mathcal{C}^{\mathit{ctx}}_{\mathit{prop}} = \{\mathit{false}, \mathit{true}\}^{\lvert \mathcal{C}^{\star}_{\mathit{prop}} \rvert}$	Properties of nodes in the hierarchical backbone
Parameterized abstract domain	**Description**
$l^{O\sharp} \in \mathit{Loc}^{O\sharp} = \mathit{Loc}_{\mathit{prop}} \times \mathcal{T}$	Abstract objects
$l^{C\sharp} \in \mathit{Loc}^{C\sharp} = \mathit{Loc}_{\mathit{prop}} \times \mathcal{T} \times \mu\mathcal{H}^{\mathit{Digest}}_{\mathit{prop}}$	Abstract child-ports
$l^{\sharp} \in \mathit{Loc}^{\sharp} = \mathit{Loc}^{O\sharp} \cup \mathit{Loc}^{C\sharp}$	Abstract locations
$v^{\sharp} \in \mathit{Val}^{\sharp} = \mathit{Loc}^{\sharp} \cup \{\mathsf{null}\} \cup \{\mathit{true}, \mathit{false}\}$	Abstract values
$\mathit{ip}^{\sharp} \in \mathcal{IP}^{\sharp} = \{\mathit{Loc}^{\sharp^{n}} \mid 0 \leq n\}$	Abstract microheap in-ports
$h^{\sharp} \in \mathcal{H}^{\sharp} = 2^{\mathit{Loc}^{O\sharp} \times \mathcal{F} \times \mathit{Val}^{\sharp}}$	Abstract intra-microheap link structure
$\mu h^{\sharp} \in \mu\mathcal{H}^{\sharp} = \mathcal{IP}^{\sharp} \times 2^{\mathit{Loc}^{O\sharp}} \times 2^{\mathit{Loc}^{C\sharp}} \times 2^{\mathit{Loc}^{\sharp}} \times \mathcal{H}^{\sharp}$	Abstract microheaps
$c^{\sharp}_{\circ} \in \mathcal{C}^{\sharp}_{\circ} = \mathcal{C}^{\mathit{ctx}}_{\mathit{prop}} \times \mu\mathcal{H}^{\mathit{Digest}}_{\mathit{prop}} \times \mu\mathcal{H}^{\sharp}$	Abstract transparent components
$c^{\sharp}_{\bullet} \in \mathcal{C}^{\sharp}_{\bullet} = \mathcal{C}^{\mathit{ctx}}_{\mathit{prop}} \times \mu\mathcal{H}^{\mathit{Digest}}_{\mathit{prop}} \times \mathcal{C}^{\mathit{ctx}}_{\mathit{prop}}$	Abstract opaque components
$c^{\sharp} \in \mathcal{C}^{\sharp} = \mathcal{C}^{\sharp}_{\circ} \cup \mathcal{C}^{\sharp}_{\bullet}$	Abstract components
$e^{\sharp} \in \mathcal{E}^{\sharp} = 2^{\mathcal{C}^{\sharp}_{\circ} \times \mathit{Loc}^{C\sharp} \times \mathcal{C}^{\sharp}}$	Abstract hierarchical backbone
${}_{\scriptscriptstyle M}h^{\sharp} \in \mathcal{HH}^{\sharp} = \mathcal{C}^{\sharp}_{\circ} \times 2^{\mathcal{C}^{\sharp}_{\circ}} \times 2^{\mathcal{C}^{\sharp}_{\bullet}} \times \mathcal{E}^{\sharp}$	Abstract Hierarchical heaps
$\rho^{\sharp} \in \mathit{Env}^{\sharp} = \mathit{VarId} \rightharpoonup \{\mathit{true}, \mathit{false}\}$	Environment for non-pointer variables
$\sigma^{\sharp} \in \Sigma^{\sharp} = \mathit{Env}^{\sharp} \times \mathcal{HH}^{\sharp}$	Abstract hierarchical memory states

Figure 17: Abstract states parameterized with $\mathit{Digest}^{\star}_{\mathit{prop}}$, $\mathit{Loc}^{\star}_{\mathit{prop}}$, and $\mathcal{C}^{\star}_{\mathit{prop}}$. $\{\mathit{false}, \mathit{true}\}^{n}$ is a boolean vector of length n.

C.3 Abstract Memory States

In this section we describe abstract memory states.

An abstract memory state $\sigma^{\sharp} = \langle \rho^{\sharp}, {}_{\scriptscriptstyle M}h^{\sharp} \rangle \in \Sigma^{\sharp}$ is comprised of an environment $\rho^{\sharp} \in \mathit{Env}^{\sharp}$ of atomic (i.e., boolean) variables and an abstract macroheap ${}_{\scriptscriptstyle M}h^{\sharp} \in \mathcal{HH}^{\sharp}$.

An abstract macroheap macroheap ${}_{\scriptscriptstyle M}h^{\sharp} = \langle c^{\sharp}_{\circ}, C^{\sharp}_{\circ}, C^{\sharp}_{\bullet}, e^{\sharp} \rangle$ is comprised of of the abstract current-component $c^{\sharp}_{\circ} \in \mathcal{C}^{\sharp}_{\circ}$; the sets $C^{\sharp}_{\circ}$ and $C^{\sharp}_{\bullet}$ of abstract transparent and opaque components, respectively; and $e^{\sharp} \in \mathcal{E}^{\sharp}$, the abstract hierarchical backbone. (Note that the abstract current-component is (necessarily) transparent.)

An abstract transparent component $c^{\sharp}_{\circ} = \langle c^{\mathit{ctx}}_{\mathit{prop}}, \mathit{digest}, \mu h^{\sharp} \rangle$ is comprised of $\mathcal{C}^{\star}_{\mathit{prop}} \in \mathcal{C}^{\sharp}_{\circ}$, the context of the component; and the $\mathit{Digest}^{\star}_{\mathit{prop}} \in \mathcal{C}^{\mathit{ctx}}_{\mathit{prop}}$, the microheap-digest, and the $\mu h^{\sharp} \in \mu\mathcal{H}^{\sharp}$, the abstract microheap, pertaining to the microheap describing its contents.

An abstract opaque component $c^{\sharp}_{\bullet} = \langle c^{\mathit{ctx}}_{\mathit{prop}}, \overline{\delta_{\mathit{digest}}}, \overline{\delta_{\mathit{ctx}}} \rangle$ is comprised of $\mathcal{C}^{\star}_{\mathit{prop}} \in \mathcal{C}^{\sharp}_{\circ}$, the context the component; and $\langle \overline{\delta_{\mathit{digest}}}, \overline{\delta_{\mathit{ctx}}} \rangle$, its frozen digest and context. Note that the abstraction of the δ function is recorded by the relation between the opaque components current and frozen properties [17].

The hierarchical backbone is a set of $\langle c^{\sharp}_{\circ}, l^{C\sharp}, c^{\sharp} \rangle$ edges between an abstract (necessarily) transparent component $c^{\sharp}_{\circ}$ to a (either opaque or transparent) component $c^{\sharp}$, labeled by the abstract child-port location $l^\sharp$.

An abstract microheap $\mu h^\sharp = \langle ip^\sharp, L^{O\sharp}, L^{C\sharp}, L^{sm\sharp}, h^\sharp \rangle$ is comprised of $ip^\sharp$, a (bound) map from in-port's indices to abstract locations and a set of abstract object-locations ($L^{O\sharp} \subseteq Loc^{O\sharp}$); abstract child-port-locations ($L^{C\sharp} \subseteq Loc^{C\sharp}$); and the recorded set of summary locations ($L^\sharp \subseteq Loc^\sharp$). The microheap-internal link structure is conservatively represented by $h^\sharp \in \mathcal{H}^\sharp$, a set of $\langle l^{O\sharp}, f, l^\sharp \rangle$ edges representing that the f-field of a location represented by the abstract object-location $l^{O\sharp} \in Loc^{O\sharp}$ *may* point to an (either object- or child-port- location $l^\sharp$). If both $l^{O\sharp}$ and $l^\sharp$ are *not* summary locations, then the recorded information is a *must point-to*. $h^\sharp$ Also records may (resp. must) information about the atomic values of fields.

An abstract value $v^\sharp \in Val^\sharp$ is either an atomic value (*null*, $true$, or $false$) or an abstract location. An abstract object location $l^{O\sharp} \in Loc^{O\sharp}$ is the pair of the location's properties and the type $type \in \mathcal{T}$ of the object at that location. An abstract child-port location $l^{C\sharp} = \langle l_{prop}, type, c^{ctx}_{prop} \rangle$ is a triplet of the location's properties, the type $type \in \mathcal{T}$ of the header of the (necessarily) sealed header of the microheap that the child-port at that location is linked to, and to its digest. If the child-port is linked to an opaque c, then we use the reserved identifier ? to describe the unknown type of the header. (Recall that opaque components carry with them the frozen digest.)

C.4 The Abstraction Function

Fig. 18 formally defines the β extraction function. As described in Sec. 3, the abstract domain, and the extraction function, are parametric in the recorded properties of: (i) locations inside microheaps, (ii) nodes and labels in the component-tree, and (iii) subcomponents recorded in their containing components.

The analysis encodes sets of properties of locations resp. microheaps resp. components utilizing the *property vectors*[3] shown in Fig. 17. We assume to be given the following property extraction functions which map locations resp. microheaps resp. components to property vectors:

- $digest(\mu h) \in \mu\mathcal{H}^{Digest}_{prop}$ maps a microheaps of sealed components to their (encoded) set of external visible properties, which we refer to as the microheap digest.
- $l_{prop}(\mu h, l) \in Loc_{prop}$ maps locations in the microheaps of sealed components to their (encoded) set of microheap-internal properties. $l_{prop}(\mu h, \rho, l) \in Loc_{prop}$ does the same for locations in the microheap of the current component.
- $c^{ctx}_{prop}(\langle \rho, \mu h, e, \Psi \rangle, c) \in \mathcal{C}^{ctx}_{prop}$ maps components to the (encoded) set of context-property, in a given environment ρ, microheap of the root component μh, a given backbone e and a map Ψ from components to their microheap digest.

The use of boolean vectors to encode the abstraction is inspired by [28].

[3] We assume every property is associated with a fixed index in the property vector.

$$\beta : \Sigma \times \overline{\Delta} \rightharpoonup \Sigma^\sharp \;\; s.t.$$

$$\beta(\langle \rho, {}_{M}h, \delta, \langle \overline{\delta_{digest}}, \overline{\delta_{ctx}} \rangle\rangle) = \langle \rho|_{\{x \in VarId : \rho(x) \notin Loc\}}, {}_{M}\beta_{\Phi,\Psi,\Theta,\overline{\Theta}}({}_{M}h)\rangle$$

where:

$$\Phi = \lambda c' \in C. l \in Loc. \begin{cases} l_{prop}(d(c'), \rho, l) & c = c' \\ l_{prop}(d(c'), l) & c \neq c' \end{cases}$$

$$\Psi = \lambda c' \in C \setminus \{c\}. \begin{cases} digest(\mu h') & \mu h' = d(e(c')(l)) \\ \overline{\delta_{digest}}(\delta(c')) & c' \notin dom(d) \end{cases}$$

$$\Theta = \lambda c' \in C.\, c^{ctx}_{prop}(\langle \rho, d(c), e, \Psi\rangle, c')$$

$$\overline{\Theta} = \lambda c' \in C \setminus dom(d).\, \overline{\delta_{ctx}}(\delta(c'))$$

(a) Memory state abstraction function. ${}_{M}h = \langle c, C, e, d\rangle$.

$${}_{M}\beta : (\mathcal{C} \rightharpoonup Loc \rightharpoonup Loc_{prop}) \times (\mathcal{C} \rightharpoonup \mu\mathcal{H}^{Digest}_{prop}) \times (\mathcal{C} \rightharpoonup \mathcal{C}^{ctx}_{prop}) \times (\mathcal{C} \rightharpoonup \mathcal{C}^{ctx}_{prop}) \times {}_{M}\mathcal{H} \rightharpoonup \mathcal{H}\mathcal{H}^\sharp \;\; s.t.$$

$${}_{M}\beta_{\Phi,\Psi,\Theta,\overline{\Theta}}(\langle c, C, e, d\rangle) = \langle {}_{C}\beta(c), C^\sharp_\circ, C^\sharp_\bullet, e^\sharp\rangle$$

where:

$$C^\sharp_\circ = \{{}_{C}\beta(c') \mid c' \in dom(d) \setminus \{c\}\}$$

$$C^\sharp_\bullet = \{{}_{C}\beta(c') \mid c' \in C \setminus dom(d)\}$$

$$e^\sharp = \{\langle {}_{C}\beta(c_s), f, {}_{C}\beta(c_t)\rangle \mid e(c_s)(f) = c_t\}$$

$${}_{C}\beta(c') = \begin{cases} \langle \Theta(c'), \Psi(c'),\; \mu\beta_{\Phi(c'),\psi \circ e(c'),\tau}(d(c'))\rangle & d(c') = \langle ip', L^{O'}, L^{C'}, h', t'\rangle \\ \quad \tau = \lambda l \in L^{C'}. \begin{cases} T(\mu h'') & \mu h'' = d(e(c')(l)) \\ ? & e(c')(l) \notin dom(d) \end{cases} & \\ \langle \Theta(c'), \Psi(c'), \overline{\Theta}(c')\rangle & c' \in C \setminus dom(d) \end{cases}$$

(b) Backbone abstraction function. ${}_{M}h = \langle c, C, e, d\rangle$. $T(\langle ip, L^O, L^C, h, t\rangle) = t(ip(1))$.

$$\mu\beta : (Loc \rightharpoonup Loc_{prop}) \times (Loc \rightharpoonup \mu\mathcal{H}^{Digest}_{prop}) \times (Loc \rightharpoonup \mathcal{T}) \times \mu\mathcal{H} \rightharpoonup \mu\mathcal{H}^\sharp \;\; s.t.$$

$$\mu\beta_{\phi,\psi,\tau}(\langle ip, L^O, L^C, h, t\rangle) = \langle ip^\sharp, L^{O\sharp}, L^{C\sharp}, L^{sm\sharp}, h^\sharp\rangle$$

where:

$$ip^\sharp = \varphi \circ ip$$

$$L^{O\sharp} = \{\varphi(l) \mid l \in L^O\}$$

$$L^{C\sharp} = \{\varphi(l) \mid l \in L^C\}$$

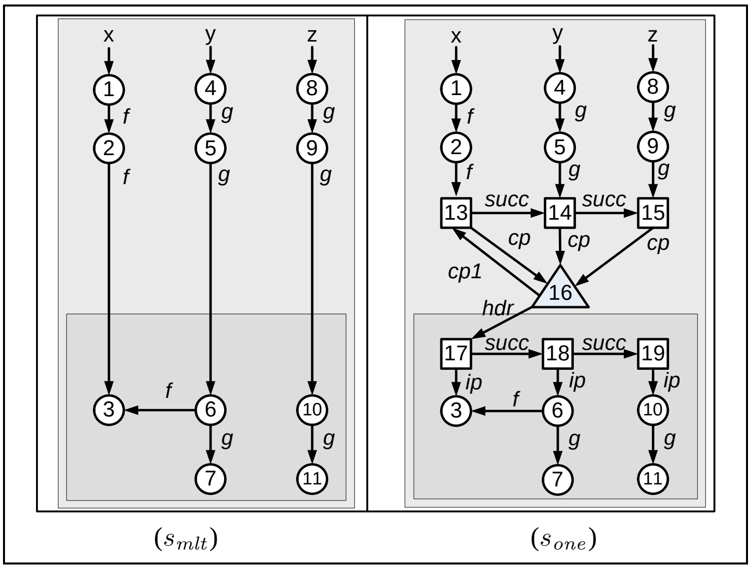
$$L^{sm\sharp} = \{l^\sharp \mid 1 < |\{l \in L^O \cup L^C : \varphi(l) = l^\sharp\}|\}$$

$$h^\sharp = \{\langle \varphi(l_s), f, \varphi(l_t)\rangle \mid h(l_s, f) = l_t \in Loc\} \cup \{\langle \varphi(l_s), f, v\rangle \mid h(l_s, f) = v \notin Loc\}$$

$$\varphi(l) = \begin{cases} \langle \phi(l), t(l)\rangle & l \in L^O \\ \langle \phi(l), \tau(l), \psi(l)\rangle & l \in L^C \end{cases}$$

(c) Microheaps abstraction function. $\mu h = \langle ip, L^O, L^C, h, t\rangle$. The environment ρ maps variable values to locations inside the microheap μh or to *null*. If μh is not the microheap of the current component, then all variables are mapped to *null*.

Figure 18: Parameterized extraction function.



(s_{mlt}) (s_{one})

Figure 19: Simulating multiple in-ports with a single in-port.

D Simulating Multiple Headers Using a Single Header

Fig. 19 shows how to simulate arbitrary number of in-ports using a single in-port. The main idea is to use correlated lists at the component and at the subcomponent. Note that we still require that the component-containment relation be a tree.

Memory state s_{mlt} depicts 2 components drawn inside shaded rectangles. There are 3 inter-component edges. Specifically, the `f`-field of the object at location 2 points to the object at location 3 and the `g`-fields of the objects at locations 5 and 9 point to the objects at locations 6 and 10, respectively. Thus, the subcomponent has 3 in-ports.

Memory state s_{one} shows how to encode memory state s_{mlt} using a single in-port. Object 16 interconnects the 2 components using 2 lists. The list nodes 13–15 are in its own component. They interpose the original references to the subcomponent. The list nodes 17–19 are in the subcomponent. They hold references to the original in-ports.

To find the target of an inter-component edge, one needs to traverse the 2 lists simultaneously. For example, to find the target of the `g`-field of object 15, one needs to follow the `cp`-edge to the interconnecting object 16; follow both the `cp1`-edge and the `hdr`-edge to get to the first nodes in both lists; traverse both lists simultaneously using interlocked steps until object 15 is reached in the list at the parent-component. At this point we have reached object 19 in the list at the child-component. Following the `ip`-edge, brings us to the target of the `g`-edge in memory state s_{mlt}.

We note that performing such an operation in the analysis may be very expensive and imprecise. Precision can be improved, however, if an a priori bound on the number of in-ports is known.