Modular Shape Analysis for Dynamically Encapsulated Programs

TAU-CS-107/06

N. Rinetzky[1] A. Poetzsch-Heffter[2] G. Ramalingam[3] M. Sagiv[1] E. Yahav[4]

[1] School of Computer Science; Tel Aviv University; Tel Aviv 69978; Israel.
{*maon,msagiv*}@*tau.ac.il*

[2] FB Informatik; TU Kaiserslautern; D-67653 Kaiserslautern; Germany.
poetzsch@informatik.uni-kl.de

[3] Microsoft Research India; Bangalore 560 080; India.
grama@microsoft.com

[4] IBM T.J. Watson Research Center; Hawthorne, NY 10532; USA.
eyahav@us.ibm.com

Abstract

We present a *modular* static analysis which identifies structural (shape) invariants for a subset of heap-manipulating programs. The subset is defined by means of a non-standard operational semantics which places certain restrictions on aliasing and sharing across modules. More specifically, we assume that live references (*i.e.*, used before set) between subheaps manipulated by different modules form a tree. We develop a conservative static analysis algorithm by abstract interpretation of our non-standard semantics. Our *modular* algorithm also ensures that the program obeys the above mentioned restrictions.

Contents

1 Introduction

Modern programs rely significantly on the use of heap-allocated linked data structures. In this paper, we present a novel method for automatically verifying properties of such programs in a modular fashion. We consider a program to be a collection of modules. We develop a shape (heap) analysis which treats each module separately. Modular analyses are attractive because they promise scalability and reuse.

Modular analysis [13], however, is particularly difficult in the presence of aliasing. The behavior of a module can depend on the aliasing created by clients of the module and vice versa. Analyzing a module making worst-case assumptions about the aliasing created by clients (or vice versa) can complicate the analysis and lead to imprecise results. Instead of analyzing arbitrary programs, we restrict our attention to certain "well-behaved" programs. The main idea behind our approach is to assume a modularly-checkable program-invariant concerning aliases of live intermodule references.

Motivating Example

Fig. 1 shows the code of a module, m_{RP}, which serves as our running example. The code is written in a Java-like language. Module m_{RP} contains two classes: Class `R` is a class of resources to be used by clients of the module. A resource has a recursive field, `n`, which is used to link resources in an internal list. (Analogous fields can be found, ***e.g.***, in the Linux kernel timers [5].) Class `RPool` is a pool of resources which stores resources using their internal list. We assume that the `n`-field is read or written only by `RPool`'s methods: `acquire`, which gets a resource out of the pool, and `release`, which stores a resource in the pool.

Typical properties we want to verify modularly are that for *any well behaved* program that uses m_{RP}, the methods of `RPool` never leak resources and never issue an acquired resource before it is released.[1] Note that these properties do not hold for arbitrary programs because of possible aliasing in the module induced by the client behavior: Consider an invocation of `p.release(r)` in a memory state in which `p` points to a non-empty resource pool. If `r` points to the head of a resource list containing more than one resource, then the tail of the list might be leaked. If, after being released into the pool that `p` points to, `r` is released into other pools, then these pools, along with the one pointed-to by `p` share (parts) of their resource lists. Note that after a shared resource is acquired from one pool, it can still be acquired from the other pools. Finally, if the resource that `r` points to is already in `p`'s pool, then `p`'s resource list becomes cyclic. A resource which is ac-

```
public class RPool {
 private R rs;
 // transferred: { e }
 public
 void release(R e){
   e.n=this.rs;
   this.rs=e;
 }
 // transferred: { }
 public R acquire(){
   R r = this.rs;
   if (r!=null) {
     this.rs=r.n;
     r.n = null; }
   else
     r = new R();
   return r;
 }}

public class R {
 R n; ... }
```

Figure 1: Module m_{RP}.

[1]Similarly, in the analysis of a client of m_{RP}, we would like to verify that the client does not use a dangling reference to a released resource. Our analysis can establish this property.

quired from a pool whose list is cyclic, stays in the pool.[2]

Given a module, and the user specification for the other modules it uses, our analysis tries to verify that the given module is "well-behaved". If this verification is unsuccessful, the analysis gives up and reports that the module may not adhere to our constraints. Otherwise, the analysis computes invariants of the given module that hold in any "well-behaved" program containing the module. A program comprised only of successfully verified modules is guaranteed to be "well-behaved".

1.1 Overview

Non-standard semantics

The basis for our approach is a *non-standard semantics* that captures the aliasing constraints mentioned above. In this paper, a module is a collection of type-definitions and procedures, and a component is a subheap. Our semantics represents the heap as an (evolving and changing) collection of (heap) *components*. Every component is comprised of objects whose types are defined in the same module. (We say that a component *belongs to* that module.) Note that multiple components belonging to the same module may co-exist. References between components belonging to different modules are allowed, however, the *internal structure* of a component can be accessed or modified only by the (procedures in the) module to which it belongs.[3] Components can be in two different states: *sealed* and *unsealed*. Sealed components represent encapsulated data returned by a module to its callers (and, hence, are expected to satisfy certain *module invariants*). In contrast, unsealed components are components that are currently being modified and may be in an unstable state.

At any point during program execution, the internal structure of only one component is "visible" and can be accessed or mutated, *i.e.*, only one unsealed component is "visible". We refer to this component as the *current component*. The only way a sealed component can be *unsealed* (permitting its internal structure to be examined and modified) is to pass it as a parameter of an appropriate intermodule procedure call so that the component becomes part of the current component for the called procedure. Our semantics requires that all parameters and the return value(s) of intermodule procedure calls must be sealed components. For brevity, we do not consider primitive values here.

Constraints

So far we have not really placed any constraints on the program. The above are standard "good modularity principles" and most programs will fit this model with minor adjustments. Before we describe the constraints we place on sharing across modules, we describe the two key issues that motivate these constraints:

1. How can we analyze a module M without using any information about the clients of M (*i.e.*, without using information about the usage context of M)?

[2] Based on a bug we found in LEDA [28] while working on an earlier version of our approach [35]. The bug (reported and fixed) was that concatenating a list to itself created a cycle.

[3] A module m can manipulate a component of a module m' by an intermodule procedure call.

2. When analyzing a client module C that makes use of another module M, how do we handle *intermodule* calls from C to M using only the analysis results for module M (*i.e.*, without analyzing module M again)?

We say that a component *owns* another component if it has a *live* reference (*i.e.*, used before set) to the other component. The most important constraint we place is that a component cannot be owned by two or more components. As a result, the heap (or the program state) may be seen as, effectively, a tree of components. Informally, this ensures that distinct components do not share (live) state. Furthermore, we require that all references to a component from its owner have the same target object. We call this object the component's *header*.[4] We refer to a program which satisfies these constraints as a *dynamically encapsulated* program. Recall that our analysis also verifies that a program is *dynamically encapsulated*.

In this paper, we require that the module dependency relation (see Sec. 2) be acyclic. This constraint simplifies our semantics (and analysis) as module reentrancy does not need to be considered: When a module is invoked *all* of its components are guaranteed to be sealed. We note that our techniques can be generalized to handle cyclic dependencies, provided that the ownership relation is required to be acyclic.

Benefits The above constraints let us deal with the two issues mentioned above in a tractable way. The restriction on sharing between components simplifies dealing with intermodule calls as they cannot have unexpected side-effects: *e.g.*, an intermodule call on one component C_1 cannot affect the state of another component C_2 that is accessible to the caller. As for the first issue, *we conservatively identify all possible input states for an intermodule call by iteratively identifying all possible sealed components that can be generated by a module*.

Specification

We now describe the extra specification a user must provide for the modular analysis. This specification consists of: (i) a *module specification* that partitions a program's types and procedures into modules; (ii) an annotation for every (public) procedure that indicates for every parameter whether it is intended to be "transferred" to the callee or not; these annotations are only considered in intermodule procedure calls. A sealed component that is passed as a *transferred* parameter of an intermodule call cannot be subsequently used by the calling module (*e.g.*, to be passed as a parameter for a subsequent intermodule call). This constraint serves to directly enforce the requirement that the heap be a tree of components. For example, for `release` we specify that the caller transfer ownership only of the resource parameter.

Given the above specification, our modular analysis can automatically detect the boundaries of the heap-components and (conservatively) determine whether the program satisfies the constraints described above

[4]Note the slight difference in terminology: In ownership type systems, owners are objects and do not belong to their ownership contexts. In our approach, components are the owners; the component header belongs to the component that is dominated by the header.

Abstraction

Our modular analysis is obtained as an abstract interpretation of our non-standard semantics. We use a 2-step successive abstraction. We first apply a novel *trimming abstraction* which abstracts away the contents of sealed components when analyzing a module. (Loosely speaking, only the heap structure of the current component, and the aliasing relationships between intermodule references leaving the current component, are tracked.) We then apply a *bounded* conservative abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction, we show how to *lift* existing *intra*procedural shape analyses, *e.g.*, [14,24,27], to obtain a modular shape abstraction (see Sec. 4). Our analysis is parametric in the abstraction of trimmed memory states and can use different (bounded) abstractions when analyzing different modules.

Analysis

Our static analysis is conducted in an assume-guarantee manner allowing each module to be analyzed separately. The analysis, computes a conservative representation of every possible sealed components of the analyzed module in dynamically encapsulated programs. This process, in effect, identifies structural invariants of the sealed components of the analyzed module, *i.e.*, it infers module invariants (for dynamically encapsulated programs). Technically, the module is analyzed together with its *most-general-client* using a framework for interprocedural shape analysis, *e.g.*, [16, 39].

Extensions

In this paper, we use a very conservative abstraction of sealed components and inter-component references (for simplicity). The abstraction, in effect, retains no information about the state of a sealed component (which typically belongs to other modules used by the analyzed module). This can lead to an undesirable loss in precision in the analysis (in general). We can refine the abstraction by using *component-digests* [38], which encode (hierarchical) properties of whole *components* in a typestate-like manner [42]. This, *e.g.*, can allow our analysis to distinguish between a reference to a pool of closed socket components from a reference to a pool of connected socket components.

1.2 Main Contributions

(i) We introduce an interesting class of dynamically encapsulated programs; (ii) We define a natural notion of *module invariant for dynamically encapsulated programs*; (iii) We show how to utilize dynamic encapsulation to enable modular shape analysis; and (iv) We present a modular shape analysis algorithm which (conservatively) verifies that a program is dynamically encapsulated and identifies its module invariants.

2 Program Model and Specification Language

Program model

We analyze imperative object-based (*i.e.*, without subtyping) programs. A program consists of a collection of procedures and a distinguished `main` procedure. The programmer can also define her own types (à la `C` structs).

Syntactic domains We assume the syntactic domains $x \in \mathcal{V}$ of variable identifiers, $f \in \mathcal{F}$ of field identifiers, $T \in \mathcal{T}$ of type identifiers, $p \in \mathcal{PID}$ of procedure identifiers, and $m \in \mathcal{M}$ of module identifiers. We assume that types, procedures, and modules have unique identifiers in every program.

Modules We denote the module that a procedure p belongs to by $m(p)$ and the module that a type identifier T belongs to by $m(T)$. A module m_1 *depends* on module m_2 if $m_1 \neq m_2$ and one of the following holds: (i) a procedure of m_1 invokes a procedure of m_2; (ii) a procedure of m_1 has a local variable whose type belongs to m_2; or (iii) a type of m_1 has a field whose type belongs to m_2.

Procedures A procedure p has local variables (V_p) and formal parameters (F_p), which are considered to be local variables, *i.e.*, $F_p \subseteq V_p$. Only local variables are allowed.

Specification language

We expect to be given a partitioning of the program types and procedures into modules. Every procedure should have an ownership transfer specification given by a set $F_p^t \subseteq F_p$ of *transferred (formal) parameters*. (A formal parameter is a transferred parameter if it points to a transferred component in an intermodule call.) For example, `e` is `release`'s only transferred parameter, and `acquire` has none.

Simplifying assumptions

We assume that procedure invocations should be *cutpoint-free* [39]. (We explain this assumption, and a possible relaxation, in Sec. 3.2.2.) In addition, to simplify the presentation, we make the following assumptions: (a) A program manipulates only pointer-valued fields and variables; (b) Formal parameters *cannot* be assigned to; (c) Objects of type T can be allocated and references to such objects can be *used as l-values* by a procedure p only if $m(p) = m(T)$; (d) Actual parameters to an intermodule procedure call should not be aliased and should point to a component owned by the caller. In particular, they should have a non-*null* value; and (e) The caller always becomes the owner of the return value of an intermodule procedure call.

3 Concrete Dynamic-Ownership Semantics

In this section, we define $\mathcal{DOS}$, a non-standard semantics which checks whether a program executes in conformance with the constraints imposed by the dynamic encapsulation model. ($\mathcal{DOS}$ stands for *dynamic-ownership semantics*.) $\mathcal{DOS}$ provides the execution traces that are the foundation of our analysis. For space reasons, we only discuss key aspects of the operational semantics, formally defined in App. D.

$\mathcal{DOS}$ is a *store-based* semantics (see, *e.g.*, [34]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{DOS}$, on the other hand, is a *local heap* semantics [36]: A memory state which occurs during the execution of a procedure does not represent objects which, at the time of the invocation, were not reachable from the actual parameters.

$\mathcal{DOS}$ is a small-step operational semantics [32]. Instead of encoding a stack of activation records inside the memory state, as traditionally done, $\mathcal{DOS}$ maintains a *stack of program states* (see App. Band [21]): Every program state contains a program point and a memory state. The program state of the *current procedure* is stored at the top of the stack, and it is the only one which can be manipulated by intraprocedural statements. When a procedure is invoked, the *entry memory state* of the callee is computed by a $Call$ operation according to the caller's current memory state, and pushed into the stack. When a procedure returns, the stack is popped, and the caller's *return memory state* is updated using a Ret operation according to its memory state before the invocation (the *call memory state*) and the callee's (popped) *exit memory state*.

The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the local heap of just one procedure. An execution trace of a program P always begins with P's `main` procedure starts executing on an *initial memory state* in which all variables have a *null* value and the heap is empty. We say that a memory state is *reachable* in a program P if it occurs as the current memory state in an execution trace of P.

3.1 Memory States

Fig. 2 defines the concrete semantic domains and the meta-variables ranging over them. We assume Loc to be an unbounded set of locations. A value $v \in Val$ is either a location, *null*, or $\ominus$, the inaccessible value used to represent references which should not be accessed.

A memory state in the $\mathcal{DOS}$ semantics is a 5-tuple $\sigma = \langle \rho, L, h, t, m \rangle$. The first four components comprise, essentially, a 2-level store: $\rho \in \mathcal{E}$ is an environment assigning values for the variables of the *current* procedure. $L \subset Loc$ contains the locations of allocated objects. (An object is identified by its location. We interchangeably use the terms object and location.) $h \in \mathcal{H}$ assigns values to fields of allocated objects. $t \in \mathcal{TM}$ maps every allocated object to the type-identifier of its (immutable) type. Implicitly, t associates every allocated location to a module:

$$\begin{array}{|l|}
\hline
l \in Loc \\
v \in Val = Loc \cup \{null\} \cup \{\ominus\} \\
\hline
\rho \in \mathcal{E} = \mathcal{V} \hookrightarrow Val \\
h \in \mathcal{H} = Loc \hookrightarrow \mathcal{F} \hookrightarrow Val \\
t \in \mathcal{TM} = Loc \hookrightarrow \mathcal{T} \\
\sigma \in \Sigma = \mathcal{E} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M} \\
\hline
\end{array}$$

Figure 2: Semantic domains.

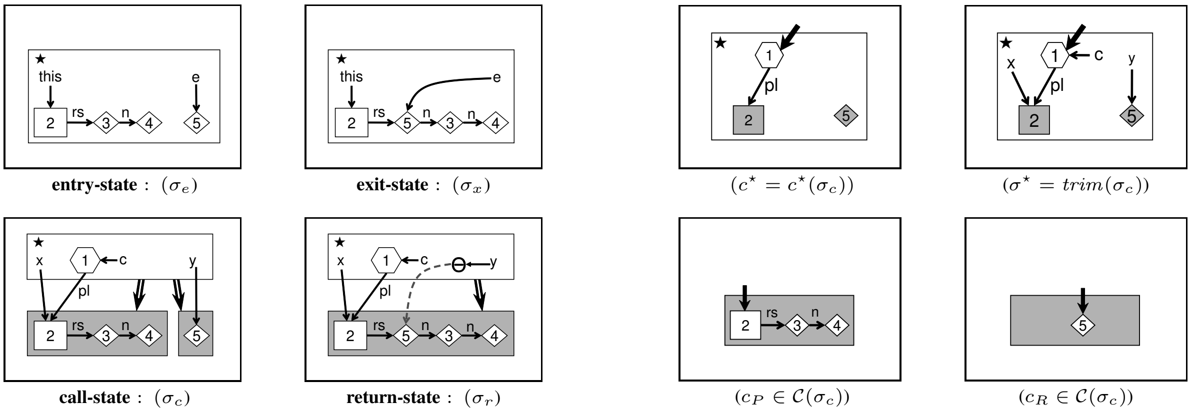


Figure 3: $(\sigma_c, \sigma_e, \sigma_x, \sigma_r)$: $\mathcal{DOS}$ memory states occurring in an invocation of `x.release(y)` on σ_c. $(c^\star, c_P, c_R)$: The implicit components of σ_c. $(\sigma^\star)$: The trimmed memory state induced by σ_c.

The module that a location $l \in L$ *belongs to in memory state* σ, denoted by $m(t(l))$, is $m(t(l))$. The additional component, $m \in \mathcal{M}$, is the module of the current procedure. We refer to m as the *current module* of σ. (We denote the current module of a state σ by $m(\sigma)$.)

Note that in $\mathcal{DOS}$, reachability, and thus domination,[5] are defined with respect to the *accessible heap*, *i.e.*, $\ominus$-valued references do not lead to any object.

> **Example 3.1** Fig. 3 (σ_c) depicts a possible $\mathcal{DOS}$ memory state that may arise in the execution of a program using the module m_{RP}. The state contains a *client* object (shown as an hexagon) pointed-to by variable `c` and having a `pl`-field pointing to a resource pool (shown as a rectangle). The resource pool, containing two resources (shown as diamonds) is also pointed-to by a variable `x`. In addition, a local variable `y` points to a resource outside the pool. (The numbers attached to nodes indicate the location of objects. The value of a (non-null) pointer variable is shown as an edge from a label consisting of the variable name to the object pointed-to by the variable. *null*-valued variables are not shown in the figure. The value of a (non-null) field f of an object is shown as an f-labeled edge emanating from the object. The lack of such an edge indicates that the field has a *null* value. Other graphical elements can be ignored for now.) The states σ_c and σ_e (also shown in Fig. 3), depict, respectively, the call- and the entry-memory states of an invocation of `x.release(y)` which we use as an example throughout this section. Note that σ_e represents only the values of the local variables of `release` and does not represent the

[5] An object l_2 is *reachable from* (resp. *connected to*) an object l_1 in a memory state σ if there is a directed (resp. undirected) path in the heap of σ from l_1 to l_2. An object l is *reachable* in σ if it is reachable from a location which is pointed-to by some variable. An object l is a *dominator* if every access path pointing to an object reachable from l, must traverse through l.

(unreachable) client-object. In the return memory state of the invocation, depicted in Fig. 3 (σ_r), the dangling reference y has the $\ominus$-value, and the resource pool dominates the resources in its list. (The return state *does not* represent the value of y before the call, indicated by the dashed arrow.)

3.1.1 Components

Intuitively, a component provides a partial view of a $\mathcal{DOS}$ memory state σ. A component of σ consists of a set of reachable objects in σ, which all belong to the same module, and records their types, their link structure, and their *spatial interface i.e.*, references to and from immediately connected objects and variables.

More formally, a component $c \in \mathcal{C} = 2^{Loc} \times 2^{Loc} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M}$ is a 6-tuple. A *component* $c = \langle I, L, R, h, t, m \rangle$ is a *component of* a $\mathcal{DOS}$ memory state σ if the following holds: L, the set of c's *internal objects*, contains only reachable objects in σ. $I \subseteq L$ and $R \subseteq Loc \setminus L$ constitute c's spatial interface: I records the *entry locations* into c. An object inside c is an *entry location* if it is pointed-to by a variable or by a field of a *reachable* object outside c. R is c's *rim*. An object outside c is in c's rim if it is pointed-to by a field of an object inside c. h defines the values of fields for objects inside c. We refer to a field pointing to an internal resp. rim object as an intra- resp. inter-component reference. h should be the restriction of σ's heap on L. t defines the types of the objects inside c and in its rim. t should be the restriction of σ's type map on $L \cup R$. m is c's *component module*. We say that component c belongs to m. The type of every object inside c must belong to m. (If L is empty then m must be the current module of σ.) Note that a component c records (among other things) all the aliasing information available in σ pertaining to fields of c's internal objects. For reasons explained below, we treat a variable pointing to a location outside the current component as an inter-component reference leaving the current component, and add that location to its rim (and relax the definition of a component accordingly).

Example 3.2 Memory state $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c \rangle$, depicted in Fig. 3, is comprised of three components. A rectangular frame encompasses the internal objects of every component. The current component, marked with a star, belongs to m_c, the client's module. The sealed components, drawn shaded, belong to module m_{RP}. Fig. 3 ($c^\star$) depicts $c^\star = \langle I^\star, L^\star, R^\star, h^\star, t^\star, m_c \rangle$, the current component of σ_c, separately from σ_c. The client-object is the only object inside $c^\star$. It is also an entry location, *i.e.*, $I^\star = L^\star = \{1\}$. An entry location is drawn with a wide arrow pointing to it. The resource pool and the resource are rim objects, *i.e.*, $R^\star = \{2, 5\}$. Rim objects are drawn opaque. The pl-labeled edge depicts the only (inter-component) reference in $c^\star$. Note that $h^\star = h_c|_{\{1\}}$ and $t^\star = t_c|_{\{1,2,5\}}$. Fig. 3 ($c_P$) and ($c_R$) depict σ_c's sealed components.

The types of the reachable objects in a memory state σ induce a (unique) *implicit component decomposition* of σ: (i) a single *implicit current component*, denoted by $c^\star(\sigma)$, containing all the *reachable* objects in σ that belong to σ's current module and (ii) a set of *implicit sealed components*, denoted by $\mathcal{C}(\sigma)$, containing (disjoint subsets of) all the *other* reachable objects. Two objects *reside within* the same implicit sealed component if they belong to the same module $m_s \neq m(\sigma)$ and are connected in σ's heap via an *undirected heap path* which only goes through objects that belong to module m_s.

The component decomposition of a memory state σ induces an *implicit component (directed) graph*. The nodes of the graph are the implicit components of σ. The graph has an edge from c_1 to c_2 if there is a rim object in c_1 which is an entry location in c_2, *i.e.*, if there is a reference from an object in c_1 to an object in c_2. For simplicity, we assume that the graph is connected, and treat local variables in a way that ensures that.

> **Example 3.3** Component $c^\star$, c_P, and c_R are the implicit components of σ_c, *i.e.*, $c^\star = c^\star(\sigma_c)$ and $\{c_P, c_R\} = \mathcal{C}(\sigma_c)$. Double-line arrows depict the edges of the component graph. This graph is connected because $c^\star$'s rim contains the resource pointed-to by y.

From now on, whenever we refer to a component of a memory state σ, we mean an implicit component of σ, and use the term *implicit component* only for emphasis. (For formal definitions of components and of component graphs, see App. D.2.)

3.1.2 Dynamically encapsulated memory state

We define the constraints imposed on memory states by the dynamic encapsulation model by placing certain restrictions on the allowed implicit components and induced implicit component graphs.

Definition 3.4 (Dynamic encapsulation) *A $\mathcal{DOS}$ memory state $\sigma \in \Sigma$ is said to be* ***dynamically encapsulated****, if (i) the implicit component graph of σ is a directed tree and (ii) every (implicit) sealed component in σ has exactly one entry location.*

We refer to the parent (resp. child) of a component c in the component tree as the *owner* of c (resp. a subcomponent of c). We refer to the single entry location of a sealed component c in a dynamically encapsulated memory state σ as c's *header*, and denote it by $hdr(c)$. We denote the module of a component c by $m(c)$.

Invariant 1 *The following properties hold in every dynamically encapsulated $\mathcal{DOS}$ memory state $\sigma \in \Sigma$ and its implicit decomposition:*

(i) A local variable can only point to a location inside $c^\star(\sigma)$, the current component of σ, or to the header of one of $c^\star(\sigma)$'s subcomponents.
(ii) For every component, every rim object is the header of a sealed component of σ.
(iii) A field of an object in a component of σ can only point to an object inside c, or to the header of one of c's subcomponents.
(iv) All the objects in a sealed component are reachable from the component's header.
(v) A header dominates its reachable heap.[5]
(vi) Every reachable object is inside exactly one component.
(vii) All the locations in one component are of the same module. The locations in the current component belong to the current module of σ.
(viii) If $c_1 \in \mathcal{C}(\sigma)$ owns $c_2 \in \mathcal{C}(\sigma)$ then $m(c_1)$ depends *on $m(c_2)$.*

$\mathcal{DOS}$ preserves dynamic encapsulation. Thus, from now on, whenever we refer to a $\mathcal{DOS}$ memory state, we mean a *dynamically encapsulated* $\mathcal{DOS}$ memory state. As a consequence of our simplifying assumptions and the acyclicity of the module dependency relation, the following holds for every $\mathcal{DOS}$ memory state σ: (i) The internal objects of $c^\star(\sigma)$ are exactly those that the current procedure can manipulate without an (indirect) intermodule procedure call. (ii) The rim of $c^\star(\sigma)$ contains all the objects which the current procedure can pass as parameters to an intermodule procedure call.

3.2 Operational Semantics

3.2.1 Intraprocedural Statements

Intraprocedural statements are handled as usual in a two-level store semantics for pointer programs (see, *e.g.*, [34]). The only unique aspect of $\mathcal{DOS}$, formalized in App. D.3, is that it aborts if an inaccessible-valued pointer is accessed.

3.2.2 Interprocedural Statements

$\mathcal{DOS}$ is a local-heap semantics [36]: when a procedure is invoked, it starts executing on an *input heap* containing only the set of *available objects for the invocation*. An object is *available for an invocation* if it is a *parameter object*, *i.e.*, pointed-to by an actual parameter, or if it is reachable from one. We refer to a component whose header is a parameter object as a *parameter component*.

A local-heap semantics and its abstractions benefit from not having to represent unavailable objects. However, in general, the semantics needs to take special care of available objects that are pointed-to by an access path which bypasses the parameters (*cutpoints* [36]). In this paper, we do not wish to handle the problem of analyzing programs with an unbounded number of cutpoints [36], which we consider a separate research problem. Thus, for simplicity, we require that *intramodule* procedure calls should be *cutpoint-free* [39], *i.e.*, the parameter objects should dominate[5] the available objects for the invocation. (In general, we can handle a *bounded* number of cutpoints.[6])

Fig. 4 defines the meaning of the $Call$ and Ret operations pertaining to an arbitrary procedure call $y = p(x_1, \ldots, x_k)$.

3.2.3 Procedure Calls

The $Call$ operation computes the callee's *entry memory state* (σ_e). First, it checks whether the call satisfies our *simplifying* assumptions. In case of an intramodule procedure invocation, the caller's memory state (σ_c) is required to satisfy the domination condition (CPF) ensuring cutpoint-freedom. Intermodule procedure calls are invoked under even stricter conditions which are fundamental to our approach: Every parameter object must dominate the subheap reachable from it. This ensures that distinct

[6]We can treat a bounded number of cutpoints as additional parameters: Every procedure is modified to have k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, the (modified) *semantics* binds the additional parameters with references to the cutpoints. This is the essence of [16]'s treatment of cutpoints.

$$
\begin{array}{ll}
\langle Call_{y=p(x_1,\ldots,x_k)}, \sigma_c\rangle \overset{D}{\rightsquigarrow} \sigma_e & m_c = m(p) \Rightarrow \textsc{cpf}\ \boldsymbol{D}_{\rho_c,h_c}(dom(\rho_c), F_p) \\
\quad \sigma_e = \langle \rho_e, L_c, h_c|_{L_{rel}}, t_c|_{L_{rel}}, m(p)\rangle & m_c \neq m(p) \Rightarrow \textsc{dif}\ \ \forall 1 \le i < j \le k : \rho_c(x_i) \neq \rho_c(x_j) \\
\quad\quad \rho_e = [z_i \mapsto \rho_c(x_i) \mid 1 \le i \le k] & \qquad\qquad\qquad\ \ \textsc{loc}\ \ \forall 1 \le i \le k : \rho_c(x_i) \in Loc \\
\quad\quad \text{where: } L_{rel} = R_{h_c}(\{\rho_c(x_i) \in Loc \mid 1 \le i \le k\}) & \\
\langle Ret_{y=p(x_1,\ldots,x_k)}, \sigma_c, \sigma_x\rangle \overset{D}{\rightsquigarrow} \sigma_r & m_c \neq m(p) \Rightarrow \textsc{own}\ \forall z \in F_p^{nt} : \rho_x(z) \in Loc \\
\quad \sigma_r = \langle \rho_r, L_x, h_r, t_r, m_c\rangle & \qquad\qquad\qquad\ \ \textsc{dom}\ \forall z \in F_p^{nt} : \boldsymbol{D}_{\rho_x^{\ominus},h_x}(F_p^{nt}, \{z\}) \\
\quad\quad \rho_r = (block \circ \rho_c)[y \mapsto \rho_x(ret)] & \\
\quad\quad h_r = (block \circ h_c|_{L_c \setminus L_{rel}}) \cup h_x & \\
\quad\quad t_r = t_c|_{L_c \setminus L_{rel}} \cup t_x & \\
\quad\quad \text{where: } L_{rel} = R_{h_c}(\{\rho_c(x_i) \in Loc \mid 1 \le i \le k\}) & \\
\quad\quad\quad\quad \rho_x^{\ominus} = \rho_x[z \mapsto \ominus \mid m_c \neq m(p), z \in F_p^t] & \\
\quad\quad\quad\quad block = \lambda v \in Val. \begin{cases} \rho_x^{\ominus}(z_i) & v = \rho_c(x_i), 1 \le i \le k \\ v & \text{otherwise} \end{cases} &
\end{array}
$$

Figure 4: $Call$ an Ret operations for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ assuming p's formal variables are $z_1, \ldots, z_k$. $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c\rangle$. $\sigma_x = \langle \rho_x, L_x, h_x, t_x, m_x\rangle$. $F_p^{nt} = \{ret\} \cup (F_p \setminus F_p^t)$. Variable `ret` is used to communicate the return value. We use the following functions and relations, formally defined in App. D.3: $R_h(L)$ computes the locations which are reachable in heap h from the set of locations L. The auxiliary relation $\boldsymbol{D}_{\rho,h}(V_I, V_D)$ holds if the set of objects pointed-to by a variable in V_D, according to environment ρ, dominates the part of heap h reachable from them, with respect to the objects pointed-to by the variables in V_I.

components are unshared. However, there is no need to check these conditions as they are invariants in our semantics: Inv. 1(i,iv,v) ensures that every parameter object to an intermodule procedure call is a header which dominates its reachable heap. (Note that Inv. 1(iv) can be exploited to check whether an object is a dominator by only inspecting access paths traversing through its component.) Thus, only our simplifying assumptions pertaining to non-nullness (LOC) and non-aliasing of parameters (DIF) need to be checked.

The entry memory state is computed by binding the values of the formal parameters in the callee's environment to the values of the corresponding actual parameters; projecting the caller's heap and type map on the available objects for the invocation; and setting the module of the entry memory state to be the module of the invoked procedure.

Note that in intermodule procedure calls, the change of the current module implicitly changes the component tree: all the available objects for the invocation which belong to the callee's module constitute the callee's current component. By Inv. 1 (vi,viii), these objects must come from parameter components.

Example 3.5 Fig. 3 (σ_e) shows the entry memory state resulting from applying the $Call$ operation pertaining to the procedure call `x.release(y)` on the call memory state σ_c, also shown in Fig. 3. All the objects in σ_e belong to m_{RP}, and thus, to its current component. Note that the latter is, essentially, a fusion of c_P and c_R, the sealed components in σ_c.

Note: The current component of a $\mathcal{DOS}$ memory state $\sigma \in \Sigma$ is the root of the component tree induced by the *local heap* represented in σ. In a *global heap*, this current component might have been one or more non-root subcomponents of a larger component-tree which is only partially visible to the current procedure. For example, the current component of the client procedure is not visible during the execution of `release`.

3.2.4 Procedure Returns

The caller's return memory state (σ_r) is computed by a Ret operation. When an *intermodule* procedure invocation returns, Ret first checks that in the exit memory state (σ_x) every non-transferred formal parameter points to an object ($\textsc{own}$) which dominates its reachable subheap ($\textsc{dom}$). This ensures that returned components are disjoint and, in particular, that the procedure's execution respected its ownership transfer specification. (Here we exploit simplifying assumption (b) of Sec. 2.)

Ret updates the caller's memory state (which reflects the program's state at the time of the call) by carving out the input heap passed to the callee from the caller's heap and replacing it instead with the callee's (possibly) mutated heap. In $\mathcal{DOS}$, an object never changes its location and locations are never reallocated. Thus, any pointer to an available object in the caller's memory state (either by a field of an unavailable object or a variable) points after the replacement to an up-to-date version of the object.

Most importantly, the semantics ensures that any future attempt by the caller to access a transferred component is foiled: We say that a local variable of the caller is *dangling* if, at the time of the invocation, it points to (the header of) a component transferred to the callee. A pointer field of an object in the caller's memory state which was unavailable for the invocation is considered to be *dangling* under the same condition. The semantics enforces the transfer of ownership by *blocking*: assigning the special value $\ominus$ to every dangling reference in the caller's memory state. (Blocking also occurs when an *intra*module procedure invocation returns to propagate ownership transfers done by the callee.) Note that cutpoint-freedom ensures that the only object that separate the callee's heap from the caller's heap are parameter objects. Thus, in particular, the only references that might be blocked point to parameter objects.

When an intermodule call returns, and the current module changes, the component tree is changed too: The callee's current component may be split into different components whose headers are the parameter objects pointed-to by non-transferred parameters. These components may be different from the (input) parameter components.

Example 3.6 Fig. 3 (σ_r) depicts the memory state resulting from applying the Ret operation pertaining to the procedure call `x.release(y)` on the memory state σ_c and σ_x, also shown in Fig. 3. The insertion of the resource pointed-to by `y` at the call-site into the pool has (implicitly) fused the two m_{RP}-components. By the standard semantics, `y` should point to the first resource in the list (as indicated by the dashed arrow). This would violate dynamic encapsulation. $\mathcal{DOS}$, however, utilizes the *ownership specification* to block `y` thus preserving dynamic encapsulation.

3.3 Observational Soundness

We say that two values are *comparable* in $\mathcal{DOS}$ if neither one is $\ominus$. We say that a $\mathcal{DOS}$ memory state σ is *observationally sound* with respect to a standard semantics σ_G if every pair of access paths that have comparable values in σ, has equal values in σ iff they have equal values in σ_G. $\mathcal{DOS}$ *simulates* the standard 2-level store semantics: Executing the same sequence of statements in the $\mathcal{DOS}$ semantics and in the standard semantics either results in a $\mathcal{DOS}$ memory states which is observationally sound with respect to the resulting standard memory state, or the $\mathcal{DOS}$ execution gets *stuck* due to a constraint breach (detected by $\mathcal{DOS}$). A program is *dynamically encapsulated* if it does not have have an execution trace which gets stuck. (Note that the initial state of an execution in $\mathcal{DOS}$ is observationally sound with respect to its standard counterpart). (For formal definitions, see App. D.4.)

Our goal is to detect structural invariants that are true according to the *standard semantics*. $\mathcal{DOS}$ acts like the standard semantics as long as the program's execution satisfies certain constraints. $\mathcal{DOS}$ enforces these restrictions by blocking references that a program should not access. Similarly, our analysis reports an invariant concerning equality of access paths only when these access paths have comparable values.

An invariant concerning equality of access paths in $\mathcal{DOS}$ for a dynamically encapsulated program is also an invariant in the standard semantics. This makes abstract interpretation algorithms of $\mathcal{DOS}$ suitable for verifying data structure invariants, for detecting memory error violations, and for performing compile-time garbage collection.

4 Modular Analysis

This section presents a conservative static analysis which identifies conservative *module invariants*. These invariants are true in *any* program according to the $\mathcal{DOS}$ semantics and in *any dynamically encapsulated* programs according to the standard semantics.

The analysis is derived by two (successive) abstractions of the $\mathcal{DOS}$ semantics: The *trimming semantics* provides the basis of our *modular* analysis by representing only components of the analyzed module. The *abstract trimming semantics* allows for an effective analysis by providing a *bounded* abstraction of trimmed memory states (utilizing existing *intra*procedural abstractions).

Module Invariants

A *module invariant* of a module m is a property that holds for all the components that belong to m when they are not being used (*i.e.*, for sealed components). Our analysis finds module invariants by computing a conservative description of the set of all possible sealed components of the module. More formally, the *module invariant of module m for type T*, denoted by $[\![Inv_m\ T]\!] \subseteq 2^{\mathcal{C}}$, is a set of sealed components of module m whose header is of type T: a sealed component c is in $[\![Inv_m\ T]\!]$ iff there exists a reachable $\mathcal{DOS}$ memory state σ in some program such that $c \in \mathcal{C}(\sigma)$.

For example, the module invariant of module m_{RP} for type `RPoll` in our running example is the set containing all resource pools with a (possibly empty) *acyclic* finite list of resources. The module invariant of module m_{RP} for type `R` is the singleton set containing a single resource with a *nullified* `n`-field: An acquired resource always has a *null*-valued `n`-field and a released resource is inaccessible.

Trimming semantics

The trimming semantics represents only the parts of the heap which belong to the current module. In particular, it abstracts away all information contained in sealed components and the shape of the component tree.

More formally, the *domain of trimmed states* is $\Sigma^\star = \mathcal{E} \times \mathcal{C}$. The *trimmed state induced by a $\mathcal{DOS}$ memory state $\sigma \in \Sigma$*, denoted by $trim(\sigma)$, is $\langle \rho, c^\star(\sigma) \rangle$. (For example, Fig. 3 ($\sigma^\star$) depicts the trimmed memory state induced by the $\mathcal{DOS}$ memory state shown in Fig. 3 (σ_c).) We say that two trimmed memory states are *isomorphic*, denoted by $\sigma_1^\star \sim \sigma_2^\star$, if $\sigma_1^\star$ can be obtained from $\sigma_2^\star$ by a consistent location renaming. A trimmed memory state $\sigma^\star$ *abstracts* a $\mathcal{DOS}$ memory state σ if $\sigma^\star \sim trim(\sigma)$.

A trimmed memory state contains enough information to determine the induced effect [11] under the trimming abstraction of intraprocedural statements and intramodule $Call$ and Ret operations by applying the statement to *any* memory state it represents. Intuitively, the reason for this uniform behavior is that the aforementioned statements are indifferent to the *contents* of sealed components: They only consider the values of fields of objects inside the current component (inter-component references included).

Analyzing intermodule procedure calls The main challenge lies in the handling of intermodule procedure calls: Applying the induced effect of $Call$ is challenging because the *most important* information required to determine the input heap of an intermodule call is the contents of parameter components. However, this is exactly the information lost under the *trimming abstraction* of the call memory state. Applying the induced effect of Ret operations pertaining to intermodule procedure calls is challenging as it considers information about the contents of heap parts manipulated by *different* modules. (Informally, Ret is aware of the *implementation* details of these modules).

We overcome the challenge pertaining to $Call$ operations by utilizing the fact that $\mathcal{DOS}$ always changes components as a whole, *i.e.*, there is no sharing between components, thus changes to one component cannot affect *a part* of the internal structure of another component. In particular, we are *anticipating the possible entry memory states of an intermodule procedure call*: In the $\mathcal{DOS}$ semantics, the current component of an entry memory state to an intermodule procedure call is comprised, essentially, as a *necessarily* disjoint union of parameter components. Note that components are sealed only when an intermodule procedure call returns. Furthermore, the only way a sealed component can be mutated is to pass it back as a parameter to a procedure of its own module. Thus, a partial view of the execution trace, which considers only the executions of procedures that belong to the analyzed module, and collects the sealed components generated when an intermodule procedure invocation returns, can (conservatively) anticipate the possible input states for the next intermodule invocations.

Specifically, *only* a combination of *already generated sealed components* of the module can be the component parameters in an intermodule procedure invocation.

We resolve Ret's need to consider components belonging to different modules utilizing the ownership transfer specification and the limited effect of intermodule procedure invocations on the caller's current component: The only effect an intermodule procedure call has on the current component of the caller is that (i) dangling references are blocked and (ii) the return value is assigned to a local variable. (By our simplifying assumptions, the return value must point either to a parameter object or to a component not previously owned by the caller. The latter case amounts to a new object in the rim of the caller's current component). Given a sound ownership specification for the invoked procedures we can apply this effect directly to the caller's memory state. This approach can be generalized (and made more precise) to handle richer specifications concerning, *e.g.*, nullness of parameters, aliasing of parameters (and return values), and digests.

Abstract trimming semantics

We provide an effective conservative abstract interpretation [11] algorithm which determines module invariants by devising a bounded abstraction of trimmed memory states. Rather than providing a new intraprocedural abstraction and analyses, we show how to *lift* existing *intra*procedural shape analyses to obtain a modular shape abstraction. An abstraction of a trimmed memory state, being comprised of an environment of a single procedure and a subheap, is very similar to an abstraction of a standard two-level store. The additional elements that the abstraction needs to track is a bounded number of entry-locations and a distinction between internal objects and rim objects. In addition, the abstract domain, expected to support operations pertaining to basic pointer manipulating statements, should be extended to allow for: checking if a $\ominus$-valued reference is accessed; the operations required for cutpoint-free local-heap analysis: carving out subheaps reachable from variables and combining disjoint subheaps; and the ability to answer queries regarding domination by variables. The only additional operation required to implement our analysis is of *blocking*, *i.e.*, setting the values of all reference pointing to a given variable-pointed object to $\ominus$. The abstract domains of [14, 24, 27], which already support the operations required for performing standard local-heap cutpoint-free analysis, can be extended with these operations.

Modular analysis

We conduct our modular static analysis by performing an interprocedural analysis of a module together with its *most-general-client*. The most-general-client simulates the behavior of an arbitrary dynamically encapsulated (*well behaved*) client. Essentially, it is a collection of non-deterministic procedures that execute arbitrary sequences of procedure calls to the analyzed module. The parameters passed to these calls also result from an arbitrary (possibly recursive) sequence of procedure calls. The most-general client exploits the fact that *different components are effectively disjoint* to separately create the value of every parameter passed to an intermodule procedure call. Thus, any conservative interprocedural analysis of the most-general client (which uses an extended abstract domain, as discussed above, and utilizes ownership specification to determine the effect of intermodule procedure calls made by the analyzed module) can modularly detect module invariants. In particular, the analysis can be performed by extending existing interprocedural frameworks for interprocedural shape analysis, *e.g.*, [16, 39]. Note that during the analysis process we also find conservative *module implementation invariants*: Properties that hold for all possible current components at different program points inside the component in every possible execution. App. G.1 provides a scheme for constructing the most-general-client of a module. (App. G.2 also provides a characterization of the module invariants based on a fixpoint equation system).

5 Related Work

A distinguishing aspect of our work is that we integrate a shape analysis with encapsulation constraints. Our work presents a nice interplay between encapsulation and modular shape analysis: it uses dynamic encapsulation to enable modular shape analysis, and uses shape analysis to determine that the program is dynamically encapsulated. In this section, we review some closely related work to both aspects of our approach.

Modular static analysis [13] describes the fundamental techniques for modular static program analysis. These techniques allow to compose separate analyses of different program parts which detect part-local properties together with a global analysis which detects global program properties. A modular shape analysis, mainly interested in properties of the global heap, is at risk of degenerating into a whole program analysis. Nevertheless, we are able to achieve modularity by associating (an unbounded number of) different *parts of the heap* (components) with different modules. By establishing rigid spatial interfaces between parts (components) belonging to different modules and requiring that heap be, effectively, a tree of components, our analysis eliminates the need to consider intermodule aliasing. This allows us to consider properties of components as module-local and directly use the techniques of [13].

Our analysis uses, in different ways, all of the techniques described in [13]: We use *user provided interfaces* to prevent live intermodule sharing and to communicate the (limited) effect of mutations done by different modules. Our definition of a component ensures that different components never share parts of the heap and only components headers can be passed as parameters to intermodule procedure calls. The latter ensures that intermodule procedures are always passed whole components as parameters, and thus, prevents intermodule procedure calls from having side-effects on components not passed as parameters. Dynamically encapsulated memory states satisfy the above two restrictions. Our analysis utilizes this fact to *simplify* the separate analysis of modules by representing only those parts of the heap which are relevant for the analyzed module. Furthermore, we can disregard possible worst case assumptions regarding aliasing in favor of certain benign disjointness assumptions. The latter, allows us to perform a *symbolic relational* analysis in which every procedure is abstracted as a relation between input parameters to output parameters. However, we do make *worst-case assumptions* regarding the possible calling sequences of intermodule procedure calls.

An analysis which conservatively abstract the results of such calling sequences can find all possible heap components of a module because it can treat heap components as atomic values.

Modular heap analysis [25] presents a modular analysis which infers class invariants based on an abstraction of program traces. [26] is an extension which handles subtyping. The determined invariants concern values of atomic fields of objects of the analyzed class. Properties of subobjects can be also detected provided that these subobjects are encapsulated inside the state of their containing (analyzed) objects and never leak to the context. (A subobject is leaked if it was passed as a parameter from the context or as a return value, *e.g.*, in our running example resources are leaked from resource pools.) [1] modularly determines invariants regarding the value of an integer field and the length of an array field of the *same* object. Our analysis, computes shape invariants of subheaps comprised of objects that may be passed as parameters

Interprocedural shape analysis [22, 43] utilize user-specified pre- and post- conditions to achieve modular shape analysis which can handle a bounded number of flat set-like data structures. It allows objects to be placed in multiple sets. In our approach, an object can be placed only in a single separately-analyzed but arbitrarily-nested set. Other interprocedural shape analysis algorithms *e.g.*, [9, 16–18, 36, 39], compute procedure summaries, but are not modular. [17] tracks properties of single objects. The other algorithms abstract whole local heaps. Our abstraction, on the other hand, represent only a part of the local heap (*i.e.*, only the current component). This suggests possible benefits both in performance and in reuse. We note that the aforementioned approaches do not require a user specification, which we require.

Encapsulation Deep ownership models structure the heap into a tree of so-called *owner contexts*. Many contributions to the field (see [30] for a survey) use type systems to enforce the structure. Based on the structure, confinement [4] and synchronization properties are defined and guaranteed [6]. Our module-induced decomposition of a memory state into a tree of components is similar to the package-induced partitioning of a memory state into a tree of memory-regions in [44]. Our constraints are similar to external uniqueness [10], which requires that there be a *unique* reference pointing to an object from outside its (transitively) owned context. Such references can be transferred via destructive reads and borrowed within a program scope where their uniqueness may not hold. References going out of components into ancestor owners are allowed. We use shape analysis to establish dynamic encapsulation which allows multiple references from an owner component to the header of an owned component. Our ownership specification is also in the spirit of [10]'s destructive reads and borrowing. An interesting opportunity is to try to combine [10] with our approach. [7] uses shape analysis to modularly verify (specified) uniqueness of a *live* reference to an *object* (which may have live references pointing to its subobjects). Our use of sealed and unsealed components is close to the use of packed and unpacked owner contexts in Boogie [2, 23]: In a packed context, class invariants have to hold. Children of packed contexts must also be packed. Modification of objects is only allowed in unpacked contexts. Whereas in our approach sealing is implicitly connected to the semantics of procedure calls, packing and unpacking has to be explicitly specified by the programmer in Boogie, which allows Boogie to handle reentrancy. The central difference between the approaches is that our techniques infer module invariants whereas Boogie verifies class invariants provided by the programmer.

Local reasoning [31] and [3] allow to modularly conduct local reasoning [34] about abstract data structures and abstract data types with inheritance, respectively. The reasoning requires user-specified resource invariants and loop invariants. Our analysis automatically infers these invariants based on an ownership transfer specification (and an instance of the bounded parametric abstraction). [3], however, allows for more sharing than in our model. Our use of rim-objects (resp. abstract sealed components) is analogous to [3]'s use of *abstract predicates*' names (resp. resource invariants).

The $\mathcal{DOS}$ semantics is implemented as a store-based semantics mainly for simplicity. Not that in the analysis, the actual location names of locations in different components do not matter. Thus the semantics could have been implemented in as a storeless semantics. Existing storeless-semantics represent either all the dynamically allocated objects [8, 19] or all the objects in the procedure's local-heap [36]. The $\mathcal{DOS}$ semantics is novel in the sense that it represents only a subset of the procedure's local-heap.

6 Conclusions

Our long term research goal is to devise precise and efficient static shape analysis algorithms which are applicable to realistic programs. We see this work as an important step towards a modular shape analysis. While the ownership model is fairly restrictive with respect to the coupling between separate components, it is very permissive about what can happen inside a single component. This model is also sufficient to express several, natural, usage constraints that arise in practice. (In particular, when accompanied with digests.) We believe that our restrictions can be relaxed to help address a larger class of programs. We plan to pursue this line of research in future work.

Acknowledgments. We are grateful for the helpful comments of T. Lev-Ami, R. Manevich, S. Rajamani, J. Reineke, G. Yorsh, and the anonymous referees of the ESOP paper [37].

References

[1] A. Aggarwal and K. Randall. Related field analysis. In *PLDI*, 2001.

[2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] G. Bierman and M. Parkinson. Separation logic and abstractions. In *POPL*, 2005.

[4] B. Bokowski and J. Vitek. Confined types. In *OOPSLA*, 1999.

[5] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'REILLY, 3rd edition, 2005.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[7] J. Boyland. Alias burying: unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, 2001.

[8] 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.

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

[10] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, New York, NY, 1979. ACM Press.

[13] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC*, 2002.

[14] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[15] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2), 2000.

[16] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

[17] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.

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

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

[20] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

[21] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.

[22] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *CC (tool demo)*, 2005.

[23] K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP*, 2006.

[24] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.

[25] F. Logozzo. Class-level modular analysis for object oriented languages. In *SAS*, 2003.

[26] F. Logozzo. Automatic inference of class invariants. In *VMCAI*, 2004.

[27] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, 2005.

[28] K. Mehlhorn and S. Naher. *LEDA, A Platform for Combinatorical and Generic Computing*. Cambridge University Press, first edition, 1999.

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

[30] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *IWACO*, 2003.

[31] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, 2004.

[32] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[33] A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.

[34] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

[35] N. Rinetzky. Interprocedural shape analysis. Master's thesis, Technion, Israel, 2001.

[36] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.

[37] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *16th European Symposium on Programming (ESOP)*, 2007.

[38] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. Componentized heap abstractions. Tech. Rep. 164, Tel Aviv University, Dec. 2006.

[39] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, 2005.

[40] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

[41] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[42] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[43] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *VMCAI*, 2006.

[44] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time java. In *RTSS*, 2004.

The following appendixes provides formal details which were omitted from the body of the paper.

- App. A provides some standard mathematical definitions regarding binary relations, equivalence relations, and sequences.
- App. B defines our technique for handling procedure calls using a stack of program states, and exemplifies it by defining our version of the standard store-based semantics for pointer programs. It also defines the notion of interprocedural execution traces.
- App. C sets certain conventions regarding the programming language and the specification language which are of use in the following sections.
- App. D formalizes the $\mathcal{DOS}$ operational semantics and some of the definitions which were given in Sec. 3 only in the intuitive level.
- App. E formalizes the notion of module invariants using interprocedural execution traces.
- App. F defines the trimming semantics which approximates $\mathcal{DOS}$ by representing only the current component of every memory state, and investigates some of its properties.
- App. G formalizes the two ways, which were described in Sec. 4, to characterize the module invariants of $\mathcal{DOS}$ using the trimming semantics. Specifically, it (i) defines the most general client of an arbitrary module and (ii) provides a scheme for deriving an equation system whose fixpoint solution is equal to the the module invariants. The latter requires explicit memory state manipulation operations, which are defined in App. F.
- Sec. H provides an example for a bounded abstraction of trimmed memory states. More specifically, it introduces a *bounded* parametric abstraction for trimmed memory states in which every module is abstracted using a (possibly) *different* set of tracked properties.
- App. I describes some possible extensions to our method.

A Notations

In this section, we provide some standard mathematical definitions regarding binary relations, equivalence relations, and sequences.

Definition A.1 (Binary relations) *A set $\tau \subseteq \Sigma \times \Sigma$ is a **binary relation** over a set Σ. The **domain** of τ, denoted by, $dom(\tau)$, is $dom(\tau) = \{\sigma \mid \langle \sigma, \sigma' \rangle \in \tau\}$. The **range** of τ, denoted by, $range(\tau)$, is $range(\tau) = \{\sigma' \mid \langle \sigma, \sigma' \rangle \in \tau\}$. The **image** of a set $S \subseteq \Sigma$ under τ, denoted by $\tau(S)$ is $\tau(S) = \{\sigma' \in \Sigma \mid \sigma \in S,\ \langle \sigma, \sigma' \rangle \in \tau\}$. The **composition** of τ with a binary relations τ' over Σ, denoted by $\tau \circ \tau'$, is $\tau \circ \tau' = \{\langle \sigma, \sigma'' \rangle \mid \langle \sigma, \sigma' \rangle \in \tau, \langle \sigma', \sigma'' \rangle \in \tau'\}$.*

By abuse of notation, we sometimes denote the image of a singleton set $S = \{\sigma\}$ under a binary relation $\tau \subseteq \Sigma \times \Sigma$ by $\tau(\sigma)$.

Definition A.2 (Equivalence relations) *A binary relation $\approx$ over a set Σ is an **equivalence relation** if $\approx$ is* reflexive*, i.e., for every $\sigma \in \Sigma$, $\sigma \approx \sigma$;* symmetric*, i.e., for every $\sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \approx \sigma_2$ iff $\sigma_2 \approx \sigma_1$; and* transitive*, i.e., for every $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$, if $\sigma_1 \approx \sigma_2$ and $\sigma_2 \approx \sigma_3$, then $\sigma_1 \approx \sigma_3$. The **equivalence class** of $\sigma \in \Sigma$ under an equivalence class $\approx$, denoted by $[\sigma]_{\approx}$, is $[\sigma]_{\approx} = \{\sigma' \in \Sigma \mid \sigma' \approx \sigma\}$. The **quotient set** of an equivalence class $\approx$, denoted by $\Sigma/\approx$ is $\Sigma/\approx = \{[\sigma]_{\approx} \mid \sigma \in \Sigma\}$.*

Definition A.3 (Set partitioning) *A set $\mathcal{P}$ is a **partitioning** of a set S if $\mathcal{P} \subseteq 2^S$, $\{s \in P \mid P \in \mathcal{P}\} = S$ and for every $p_1, p_2 \in \mathcal{P}$ such that $p_1 \neq p_2$, $p_1 \cap p_2 = \emptyset$.*

Definition A.4 (Sequences) *A **sequence** π over a set S is a total function $\pi \in \{i \in \mathcal{N} \mid 1 \leq i \leq n\} \to S$ for some $n \in \mathcal{N}$. The **length of a sequence** π, denoted by $|\pi|$, is $|dom(\pi)|$.*

*A sequence π is a **subsequence** of a sequence π' if there exists $n \in \mathcal{N}$ such that for any $j \in dom(\pi)$ it holds that $\pi(j) = \pi'(j + n)$; such a π is (also) a **prefix** of π' if $n = 0$.*

*The **concatenation** of sequences π_1 and π_2, denoted by juxtaposition $\pi_1\pi_2$, is a sequence π such that $\pi(i) = \pi_1(i)$ for every i, $i \leq |\pi_1|$ and $\pi(i) = \pi_2(i - |\pi_1|)$ for every i, $1 \leq |\pi_1|$.*

B Interprocedural Lifting Semantics

In this section, we present a technique which allows to *lift* a concrete intraprocedural semantics into an interprocedural semantics. Specifically, we suggest a way to extend a semantics which supports only atomic (intraprocedural) statements to handle procedure invocations.

The main idea is to replace the representation of the program state used by the intraprocedural semantics by a *stack* of program states. A standard stack of activation records, contains only the local variables, the current program points (program counter), the return addresses. In contrast, the stack that we use stores *whole* program states.

Our technique requires to extend the intraprocedural semantics with operations that defines the memory state of the invoked procedure when its execution starts (according to the state of the caller at the call-site) and the memory state of the caller when it regains control (according to the caller's memory state at the call-site and the callee's memory state at the exit-site). These operations are similar to the ones used in a large-step semantics [20] to define the memory state on which the body of an invoked procedure is executed (entry memory state) and the memory state resulting after a procedure call (return memory state), see, e.g., [36].

The ideas in this section are heavily influenced by the formulation of an *interprocedural analysis* using a stack of *abstract* memory states in [21].

B.1 Procedure Representation by Flow Graphs

In the following, we assume that the bodies of procedures are represented in a standard way by their *flow graphs*. A flow graph of a procedure p is a rooted directed graph

$st \in Stmt$ ::=	$\mathtt{x} = \mathtt{alloc}\, T \mid \mathtt{x} = \mathtt{null} \mid \mathtt{x} = \mathtt{y} \mid = \mathtt{y.f} \mid \mathtt{y.f} = \mathtt{x} \mid$ $\mathtt{assume}(cnd) \mid \mathtt{y} = \mathtt{p}(\mathtt{x}_1, ..., \mathtt{x}_k)$	
$cnd \in Cond$ ::=	$\mathtt{x} = \mathtt{y} \mid \mathtt{x} \neq \mathtt{y} \mid \mathtt{x} = \mathtt{null} \mid \mathtt{x} \neq \mathtt{null}$	

Table 1: Syntax of the statements used in this paper.

$G_p = \langle N_p, E_p, s_p, e_p \rangle$.

G_p's nodes, $N_p \subset \mathcal{PP}$, are *program points*. G_p is rooted at s_p, the *entry-site* to p. The program point e_p is p's *exit-site*. Every node, except s_p (resp. e_p), is the target (resp. source) of an edge $e \in E_p$. For simplicity, we assume that the sets of program points of different procedures are disjoint.

G_p's edges, $E_p \subseteq N_p \times N_p$, are associated with *atomic* statements and *procedure calls*. Tab. 1 defines the statements used in this paper (`assume` statements are used to handle conditions).

The function $stmt_{G_p}(e)$ maps flow graph G_p edges to statements. The function out_{G_p} maps a given program point in N_p to the set of its successors[7], i.e., $out_{G_p}(n) = \{n' \in N_p \mid \langle n, n' \rangle \in E_p\}$. (We omit the G_p subscript when it is clear from the context). For simplicity, we assume that the edges emanating from s_p, as well as the ones entering into e_p, are associated with nop statements. When an edge $e = \langle n_c, n_r \rangle$ is associated with a procedure call, we say that n_c is a *call-site* and that n_r is its *corresponding return-site*. We assume every call-site n_c has exactly one return-site which we denote by $return(n_c)$. Similarly, we assume every return-site n_r has exactly one call-site, denoted by $call(n_r)$.

The function $fg(n)$ maps a program point to the (unique) flow graph which contains n. The function $fg(p)$ maps a procedure identifier p to p' flow graph. The function $proc(G)$ maps a flow graph G of procedure p to p's procedure identifer.

A program is comprised of a set of procedures, including a distinguished `main` procedure. We denote the set of all procedures in a program P by $procs(P)$, and the set of all program points in P by $PP(P)$, i.e., $PP(P) = \bigcup_{p \in procs(P)} N_p$, where N_p are the program points of procedure p.

B.2 Intraprocedural Semantics

An *intraprocedural semantics S manipulating memory states $\sigma \in \Sigma$* defines a *meaning* for every intraprocedural statement st as a binary relation over a set of memory states $[\![st]\!]_S \subseteq \Sigma \times \Sigma$. A pair of memory states $\langle \sigma, \sigma' \rangle \in [\![st]\!]_S$ (also denoted by $\sigma' \in [\![st]\!]_S(\sigma')$, see Def. A.1) iff the execution of st in memory state σ may lead to memory state σ'.

Definition B.1 (Program states) *A **program state** of a program P according to a semantics which manipulates memory states Σ is a pair comprised of a program point and a memory state, $\langle pp, \sigma \rangle \in PP(P) \times \Sigma$.*

[7]The intended meaning of several edges emanating from a single program point is that a successor is chosen non deterministically.

An intraprocedural semantics S associates to every (single-procedure) program P a *transition system* between program states $tr_P \subseteq (PP(P) \times \Sigma) \times (PP(P) \times \Sigma)$. We write $\langle pp, \sigma \rangle \overset{tr_P}{\rightsquigarrow} \langle pp', \sigma' \rangle$ for $\langle \langle pp, \sigma \rangle, \langle pp', \sigma' \rangle \rangle \in \overset{tr_P}{\rightsquigarrow}$. A transition $\langle pp, \sigma \rangle \overset{tr_P}{\rightsquigarrow} \langle pp', \sigma' \rangle$ indicates that (i) there is an edge $\langle pp, pp' \rangle \in E$, i.e., $pp' \in out(pp)$, and (ii) the execution of a statement $st = st(\langle pp, pp' \rangle)$ in memory state σ may lead to memory state σ', i.e., $\sigma' \in [\![st]\!]_S(\sigma)$.

B.3 Interprocedural Lifting

We lift an intraprocedural semantics, as defined above, to an interprocedural semantics which is capable of handling procedure calls. The main idea is that the interprocedural semantics associates with every program P a transition system between *stacks of program states* (instead of a transition system between program states).

Intuitively, the interprocedural semantics maintains a stack of program states. The program state of the *current* (active) procedure is stored at the top of the stack. Intraprocedural statements have access only to the top of the stack.

Interprocedural statements make use of the (unbounded number of) program points stored in the stack to ensure that a procedure which was invoked at a call-site n_c returns to the corresponding return-site, $return(n_c)$.

The memory states stored in the stack are used to record the memory states of the caller as it was when the control reached the invocation call-site. Our lifting technique does not dictate the memory states at the entry-sites and return-sites. Instead, it requires two operations, $Call$ and Ret, which specify, for every procedure call statement $y = p(x_1, \ldots, x_k)$, the effect of transferring control from the caller to the callee, and vice versa:

$$\begin{array}{rcl} [\![Call_{y=p(x_1,\ldots,x_k)}]\!] & \subseteq & \Sigma \times \Sigma \\ [\![Ret_{y=p(x_1,\ldots,x_k)}]\!] & \subseteq & (\Sigma \times \Sigma) \times \Sigma \end{array}$$

Intuitively, the semantics utilizes the above relations to handle procedure calls as follows:

- When a procedure is invoked, the semantics *pushes* a new program state, which we refer to as the *entry state*, to the top of the stack. The entry state is comprised of the callee's entry-site and an *entry memory state*. The entry memory state depends on the memory state at the call-site and is specified by the meaning of a *Call* operation.
- Execution of the statements in the callee's body is continued as in the case of the intraprocedural statements, manipulating the program state at the top of the stack.
- When a procedure execution reaches the exit-site, the semantics pops the callee's program state from the top of the stack. The caller's execution is continued from the return-site corresponding to the call-site stored in the stack. The memory state at the return site is updated by the Ret relation according to the call memory state and the exit memory state.

Formally, a (lifted) interprocedural semantics which is based on an intraprocedural semantics manipulating memory states $\sigma \in \Sigma$, manipulates *stacks* of program states,

$$
\begin{array}{|l|}
\hline
newstack : \mathcal{PP} \times \Sigma \to \mathcal{STK}_\Sigma \\
top : \mathcal{STK}_\Sigma \to \mathcal{PP} \times \Sigma \\
push : \mathcal{STK}_\Sigma \times \mathcal{PP} \times \Sigma \to \mathcal{STK}_\Sigma \\
pop : \mathcal{STK}_\Sigma \hookrightarrow \mathcal{STK}_\Sigma \\
|\cdot| : \mathcal{STK}_\Sigma \hookrightarrow \mathbb{N} \\
\begin{array}{rcl}
top(newstack(pp, \sigma)) & = & \langle pp, \sigma \rangle \\
top(push(stk, pp, \sigma)) & = & \langle pp, \sigma \rangle \\
pop(push(stk, pp, \sigma)) & = & stk
\end{array} \\
|stk| = \begin{cases} 1 & stk = newstack(pp, \sigma) \\ 1 + |pop(stk)| & \text{otherwise} \end{cases} \\
\hline
\end{array}
$$

Figure 5: Axiomatic definition of a stack of program states. Note that $pop(newstack(pp, \sigma))$ is undefined.

$stk \in \mathcal{STK}_\Sigma$. The equational definition of the set $\mathcal{STK}_\Sigma$ in Fig. 5 provides the following stack-manipulating operations: - $newstack$ creates a new stack containing a single program state; - $push$ pushes a program state to the top of the stack; - top retrieves the program state from the top of the stack; - pop pops the program state from the top of the stack; and - $|\cdot|$ returns the number of elements in the stack.

The use of stacks of program states allows us to formalize the notion of the *current program state*.

Definition B.2 (Current program state) *The **current program state of a stack** $stk \in \mathcal{STK}_\Sigma$ is the program state $\langle pp, \sigma \rangle = top(\sigma)$. The memory state σ, denoted by $cur_{state}(stk)$, is the **current memory state of stack** stk. The program point pp, denoted by $cur_{pc}(stk)$, is the **current program point of stack** stk. The procedure $proc(fg(pp))$, denoted by $cur_{proc}(stk)$, is the **current procedure of stack** stk.*

A (lifted) interprocedural semantics $\mathcal{S}$ associates with every program P a transition system $str \subseteq \mathcal{STK}_\Sigma \times \mathcal{STK}_\Sigma$ between *stacks* of program states. The transition system, defined in defined in Fig. 6, is parameterized by the intraprocedural transition relation, tr_P, and the meaning of $Call.$ and $Ret.$ operations. We write $stk \overset{str_P}{\rightsquigarrow} stk'$ for $\langle stk, stk' \rangle \in \overset{str_P}{\rightsquigarrow}$.

A transition $stk \overset{str_P}{\rightsquigarrow} stk'$ indicates that one of the following holds:

- $stk \overset{str_P^{Intra}}{\rightsquigarrow} stk'$: stk' results from an application of an intraprocedural statement on the current memory state of stk.
- $stk \overset{str_P^{Call}}{\rightsquigarrow} stk'$: The current program point in stk is a call-site. The entry-state which results from applying a $Call$ statement to the current memory state of stk, is pushed into stk, resulting in stk'.
- $stk \overset{str_P^{Ret}}{\rightsquigarrow} stk'$: The current program point in stk is an exit-site. The return-state which results from applying a Ret statement on the two topmost memory states in stk, the call memory state and the exit memory state, is pushed into stk, after the call state and the exit state have been popped, resulting in stk'.

$$
\begin{array}{l}
str_P \subseteq \mathcal{STK}_\Sigma \times \mathcal{STK}_\Sigma \;\; s.t. \\
str_P = str_{Intra} \cup str_{Call} \cup str_{Ret} \\
str_P^{Intra} = \\
\left\{ \langle stk, stk' \rangle \;\middle|\; \begin{array}{c} \langle pp, \sigma \rangle = top(stk), \\ \langle pp, \sigma \rangle \overset{tr_P}{\leadsto} \langle pp', \sigma' \rangle, \\ stk' = push(pop(stk), pp', \sigma') \end{array} \right\} \\
str_P^{Call} = \\
\left\{ \langle stk, stk' \rangle \;\middle|\; \begin{array}{c} \langle pp, \sigma \rangle = top(stk), \\ pp_r = return(pp), \\ st_{fg(pp)}(\langle pp, pp_r \rangle) \equiv y = p(x_1, \ldots, x_k), \\ \sigma_e \in [\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma), \\ stk' = push(stk, s_p, \sigma_e) \end{array} \right\} \\
str_P^{Ret} = \\
\left\{ \langle stk, stk' \rangle \;\middle|\; \begin{array}{c} \langle e_p, \sigma \rangle = top(stk), \\ \langle pp_c, \sigma_c \rangle = top(pop(stk)), \\ pp_r = return(pp_c), \\ st_{fg(pp_c)}(\langle pp_c, pp_r \rangle) \equiv y = p(x_1, \ldots, x_k), \\ \sigma_r \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\langle \sigma_c, \sigma \rangle), \\ stk' = push(pop(pop(stk)), pp_r, \sigma_r) \end{array} \right\}
\end{array}
$$

Figure 6: Lifted interprocedural transition system for an arbitrary program P. p is an arbitrary procedure; $fg(p) = \langle N_p, E_p, s_p, n_p \rangle$. Σ is the set of manipulated memory states.

The following definition formalizes the notion of the statement executed in a transition.

Definition B.3 (Executed statement) *The* ***executed statement in a transition*** $stk \overset{str_P}{\leadsto} stk'$*, denoted by* $stmt(\langle pp, pp' \rangle)$*, where* $pp = cur_{pc}(stk)$ *and* $pp' = cur_{pc}(stk')$*, is:*

- $st_{fg(pp)}(\langle pp, pp' \rangle)$ *if program points* pp *and* pp' *belong to the same procedure, i.e.,* $fg(pp) = fg(pp')$.
- $Call_{y=p(x_1,\ldots,x_k)}$ *if* pp *is a call-site,* $st_{fg(pp)}(\langle pp, return(pp) \rangle) \equiv y = p(x_1, \ldots, x_k)$*, and* pp' *is the entry-site of* p.
- $Ret_{y=p(x_1,\ldots,x_k)}$ *if* pp' *is a return-site,* $st_{fg(pp')}(\langle call(pp'), pp' \rangle) \equiv y = p(x_1, \ldots, x_k)$*, and* pp *is the exit-site of* p.

B.4 Interprocedural Execution Traces

In this section, we define the notion of the *reachable interprocedural traces of a program*. Based on this notion, we define the notions of *reachable memory states* and the *meaning of a procedure*. The latter is defined as an input-output relation. According to our definition, a procedure may have different meanings in different programs. However, the difference amounts to different sets of input memory states. This indicates that it is possible to define a functional meaning for a procedure as a transformer from *input states* to *output states* in a program independent manner.

In the rest of the section, we assume that P is an arbitrary program and that $\mathcal{S}$ is an arbitrary (lifted) interprocedural semantics manipulating memory states $\sigma \in \Sigma$. We denote by str_P the transition relation associated by $\mathcal{S}$ to P. We assume that the execution of every program begins at a (designated) *initial memory state*, $\sigma_0 \in \Sigma$.

Definition B.4 (Traces) *A sequence $\pi \in \Pi_{\mathcal{STK}_\Sigma}$ is a **trace** of program P if $\langle \pi(i), \pi(i+1) \rangle \in str_P$ for every $1 \leq i < |\pi|$.*

Definition B.5 (Paths) *The **path induced by a program trace** π, denoted by $path(\pi)$, is a sequence of program points $\overline{pp}$ such that $\overline{pc}(i) = cur_{pc}(\pi(i))$.*

Definition B.6 (Initial and final memory states) *The **initial resp. final memory state of a trace** π, denoted by $in(\pi)$ resp. $out(\pi)$, is the current memory state of $top(\pi(1))$ resp. $top(\pi(|\pi|))$.*

Definition B.7 (Feasible execution traces) *An execution trace π of program P is **feasible** if $\pi(1) = newstack(s_{main}, \sigma_0)$. We denote the set of feasible program execution traces of program P according to semantics $\mathcal{S}$ by $\Pi^P_{\mathcal{S}}$.*

Note: In Sec. 3, we referred, for simplicity, to every execution trace as a feasible execution trace. Here, we would like to make the distinction between traces and feasible execution traces to allow referring to suffixes of execution traces (and in particular, feasible execution traces) as execution traces too.

Definition B.8 (Reachable memory states) *Let pp be a program point in program P, $pp \in PC(P)$. A memory state $\sigma \in \Sigma$ is a **reachable memory state at** pp **(according to semantics** $\mathcal{S}$**)** if there exists a feasible execution trace $\pi \in \Pi^P_{\mathcal{S}}$ such that pp is the current program point of $\pi(|\pi|)$ and σ is the current memory state of $\pi(|\pi|)$.*

We denote the set of reachable memory states at pp in program P (according to semantics $\mathcal{S}$) by $\mathcal{R}(pp)^P_{\mathcal{S}}$.

Informally, the meaning of a program point pp of procedure p in program P is the (minimal) set of pairs of memory states $\langle \sigma_s, \sigma' \rangle$ such that if p is invoked in P with (entry) memory state σ_s, then the execution may reach program point pp—*in the same invocation of* p—with the current memory state being σ'.

The following definition allows to detect when a trace starts and ends with a program state pertaining to the same invocation of a procedure.

Definition B.9 (Complete execution traces) *A trace π of program P is a **complete execution trace in procedure p to program point** pp' if: (i) $s_p = cur_{pc}(\pi(1))$, and (ii) $pp' = cur_{pc}(\pi(|\pi|)) \in N_p$, (iii) $|\pi(1)| = |\pi(|\pi|)| \leq |\pi(i)|$ for every i, $1 \leq i \leq |\pi|$, and (iv) π is a subsequence of $\pi' \in \Pi^P_{\mathcal{S}}$. We denote the set of all complete execution traces in procedure p to program point pp in program P by $\Pi^{P,pp}_{\mathcal{S}}$.*

l	$\in$	Loc
v	$\in$	$Val_G = Loc \cup \{null\}$
ρ	$\in$	$Env = \mathcal{V} \hookrightarrow Val_G$
t	$\in$	$\mathcal{TM} = Loc \hookrightarrow \mathcal{T}$
h	$\in$	$Heap = Loc \times \mathcal{F} \hookrightarrow Val_G$
$\sigma_G, \langle \rho, L, h, t \rangle$	$\in$	$\Sigma_G = Env \times 2^{Loc} \times Heap \times \mathcal{TM}$

Figure 7: Semantic domains of the $\mathcal{GSB}$ semantics.

A complete execution trace starts at the entry of a procedure and ends in some program point in that procedure. The third requirement ensures that the memory state at the beginning of the trace and at its end belong to the same invocation of the procedure: The stack height's never goes below its height at the entry to the procedure. Furthermore, the height of the stack at the start of the trace is the equal to its height at the end of the trace. Note that the high of the stack decreases only when a Ret operation occurs. The definition of interprocedural transition relation ensures that Ret and $Call$ operations are balanced. Thus, a complete execution trace corresponds to a valid interprocedural same level path [41]. Note that a complete execution trace has to be a suffix of a feasible execution trace of P.

Definition B.10 *We say that two memory state σ_1 and σ_2 occur in the same* ***incarnation*** *of a procedure p if there exists a complete trace in procedure p for some program point $pp \in N_p$, $\pi = \pi'\pi'' \in \Pi_{\mathcal{S}}^{P,pp}$ such that (i) $|\pi''(1)| = |\pi''(|\pi''|)|$ (ii) $\sigma_1 = cur_{state}(\pi''(1))$, and (iii) $\sigma_2 = cur_{state}(\pi''(|\pi''|))$.*

Definition B.11 (Meaning of a program point) *The* ***meaning of program point*** *pp* ***of procedure*** *p* ***in program*** *P* ***(according to semantics $\mathcal{S}$)****, denoted by $[\![pp]\!]_{\mathcal{S}}^{P}$, is*

$$[\![pp]\!]_{\mathcal{S}}^{P} = \left\{ \langle in(\pi), out(\pi) \rangle \;\middle|\; \pi \in \Pi_{\mathcal{S}}^{P,pp} \right\}$$

Definition B.12 (Meaning of procedures) *The* ***meaning of procedure*** *p* ***in program*** *P, denoted by $[\![p]\!]^{P}$, is $[\![p]\!]_{\mathcal{S}}^{P} = [\![e_p]\!]_{\mathcal{S}}^{P}$.*

The following lemma indicates that the meaning of a procedure with respect to a given memory state σ in two different programs is either identical, or that in one of the programs, σ does not reach p's entry.

Lemma B.13 *Let P and P' be programs using procedure p. If $\sigma \in dom([\![p]\!]_{\mathcal{S}}^{P}) \cap dom([\![p]\!]_{\mathcal{S}}^{P'})$ then $[\![p]\!]_{\mathcal{S}}^{P}(\sigma) = [\![p]\!]_{\mathcal{S}}^{P'}(\sigma)$.*

B.5 Example: Global-Heap Store-Based Semantics

In this section, we show, as an example, how to lift the standard intraprocedural store-based semantics for heap-manipulating programs to the *standard interprocedural store-based semantics for pointer programs* ($\mathcal{GSB}$).

$\langle \mathtt{x = null}, \sigma_G \rangle \overset{G}{\rightsquigarrow} \langle \rho[x \mapsto \mathit{null}], L, h, t \rangle$	
$\langle \mathtt{x = y}, \sigma_G \rangle \overset{G}{\rightsquigarrow} \langle \rho[x \mapsto \rho(y)], L, h, t \rangle$	
$\langle \mathtt{x = y.f}, \sigma_G \rangle \overset{G}{\rightsquigarrow} \langle \rho[x \mapsto h(\rho(y), f)], L, h, t \rangle$	$(\mathbf{1})\ \rho(y) \neq \mathit{null}$
$\langle \mathtt{y.f = x}, \sigma_G \rangle \overset{G}{\rightsquigarrow} \langle \rho, L, h[(\rho(y), f) \mapsto \rho(x)], t \rangle$	$(\mathbf{1})\ \rho(y) \neq \mathit{null}$
$\langle \mathtt{x = alloc}\, T, \sigma_G \rangle \overset{G}{\rightsquigarrow} \langle \rho \cup \{l\}, L[x \mapsto l], h \cup I(l), t[l \mapsto T] \rangle$	$(\mathbf{2})\ l \notin L$
$\langle \mathtt{assume(x} \bowtie \mathtt{y)}, \sigma_G \rangle \overset{G}{\rightsquigarrow} \langle \rho, L, h, t \rangle$	$(\mathbf{3}) \rho(x) \bowtie \rho(y)$

Figure 8: Semantics of intraprocedural statements in the $\mathcal{GSB}$ semantics. $\bowtie$ stands for either $=$ or $\neq$. $\sigma_G = \langle \rho, L, h, t \rangle$. I initializes all pointer fields at l to *null*.

Fig. 7 defines the semantic domains. *Loc* is an unbounded set of memory locations. A value $v \in \mathit{Val}_G$ is either a location or *null*. A *memory state*, $\sigma_G \in \Sigma_G$, keeps track of an environment mapping the local variables *of the current procedure* to values, ρ, the allocated memory locations, L, a mapping from fields of *allocated* locations to values, h, and the types of *allocated* objects, t. Due to our simplifying assumptions, a value is either a memory location or $\mathit{null} \notin \mathit{Loc}$.

B.5.1 Operational semantics

Fig. 8 specifies a standard two-level store intraprocedural semantics for pointer programs. The semantics is specified for the statements $st \in \mathit{stms}$ defined in Tab. 1. The semantics of a statement st is described in the form of axioms. The intention is that $\langle \sigma, \sigma' \rangle \in [\![st]\!]$ iff $\langle st, \sigma \rangle \overset{G}{\rightsquigarrow} \sigma'$.

B.5.1.1 Intraprocedural operational semantics The statement `x=null` nullifies variable `x`. The statement `x=y` copies the value of variable `y` to variable `x`. These statements affect only the environment. Note that they are not constrained by a side-condition.

The axioms for field-dereferences, which observe (alteratively access or read) the heap (`x=y.f`) and for destructive-updates (alteratively modify or write), which, potentially, mutate the heap (`y.f=x`) are restricted by side-condition (1). The latter, verifies that the program does not dereference null-valued pointers.

Object allocation statements (`x = alloc T`) allocates an object of type T in an unused location l, which is assigned to variable `x`. For *simplicity*, we require that every location is allocated once during the execution of the program. This requirement is enforced by side-condition (2) and the maintenance of the set L of *all* allocated objects, including ones that are unreachable by the current procedure. Note that once an object has been allocated, it's type is immutable.

The `assume` statements are used to implement conditions. They compare the values of two variables. Side-condition (3) does the comparison.

B.5.1.2 Interprocedural operational semantics Below, we define the meaning of $Call$ and $Return$ statements pertaining to an arbitrary procedure call $y = p(x_1, \ldots, x_k)$.

$$\langle Call_{y=p(x_1,\ldots,x_k)}, \sigma_G^c \rangle \overset{G}{\rightsquigarrow} \langle L_c, [z_i \mapsto \rho_c(x_i) \mid 1 \le i \le k], h_c \rangle$$
$$\langle Ret_{y=p(x_1,\ldots,x_k)}, \sigma_G^c, \sigma_G^x \rangle \overset{G}{\rightsquigarrow} \langle L_x, \rho_c[y \mapsto \rho_x(ret)], h_x \rangle$$
$$\text{where } \sigma_G^c = \langle L_c, \rho_c, h_c \rangle \text{ and } \sigma_G^x = \langle L_x, \rho_x, h_x \rangle$$

The $Call$ operation initializes the formal parameters using the values of the actual parameters, and initiate the invoked procedure execution in a memory state that contains the caller's heap. The Ret operation copies into the return memory state the callee's heap at the exit-site, and restore the values of the caller's variables from the call memory state (stored in the stack), except for y, which is assigned the callee's return value.

Note that the semantics treats the whole heap as a single global resource. In particular, the heap part of the memory state in the call-site is not needed to define the memory state at the return-site.

C Programming Language Conventions

In this section, we set certain conventions regarding for the programming language and the specification language.

Procedures For a procedure p, V_p denotes the set of its local variables and $F_p \subseteq V_p$ denotes the set of its formal parameters. a Procedure returns a value by assigning it to a designated variable `ret`. Recall that Parameters are passed by value and that formal parameters cannot be assigned to.

Specification Language The ownership transfer specification for a procedure p is given by a set of $F_p^t \subseteq F_p$ of transferred (formal) parameters. We define the set F_p^{nt} to be the set of non-transferred (formal) parameters of procedure p. For simplicity, we assume `ret` $\in F_p^{nt}$ in case the procedure returns a value. (Recall that by our simplifying assumptions, the caller always becomes the owner of the return value). For example, $F_{release}^{nt} = \{\texttt{this}\}$ and $F_{acquire}^{nt} = \{\texttt{this}, \texttt{ret}\}$.

D Formal Details Pertaining to the $\mathcal{DOS}$ Semantics

In Sec. 3, we informally defined the (rather standard) notions of *reachability* and *connectivity*. To avoid possible misinterpretations, App. D.1 provides their formal definitions in the context of this paper.

App. D.2 formalizes the notion of implicit components and implicit component graphs. App. D.3 provides the formal definition of the operational semantics. App. D.4 provides certain properties of the $\mathcal{DOS}$ semantics.

D.1 Reachability, Connectivity, and Domination

In this section, we give formal definitions for the notions of *reachability* and (undirected) *connectivity* in $\mathcal{DOS}$ memory states. We also formalize the notion of domination. These definitions are based on the corresponding standard notions in 2-level stores. Intuitively, location l_2 is *reachable from* (resp. *connected to*) a location l_1 in a memory state σ if there is a directed (resp. undirected) path in the heap of σ from l_1 to l_2. A locations l is *reachable* in σ if it is reachable from a location which is pointed to by some variable. An object l is a *dominator* if every access path pointing to an object reachable from l, must traverse through l. Note that the inaccessible value is treated, basically, as a *null* value, *i.e.*, it cannot lead to an object.

Definition D.1 (Heap path) *A sequence π of location is a **directed heap path** in a heap $h \in \mathcal{H}$, if for every $1 \leq i < |\pi|$ there exists $f_i \in \mathcal{F}$ such that $h(\pi(i), f_i) = \pi(i+1)$. A directed heap path π **goes from** l_1, if $\pi(1) = l_1$, it **goes to** l_2 if $\pi(|\pi|) = l_2$.*

*A sequence π of location is an **undirected heap path** in $h \in \mathcal{H}$, if for every $1 \leq i < |\pi|$ there exists $f_i \in \mathcal{F}$ such that either $h(\pi(i), f_i) = \pi(i+1)$ or $h(\pi(i+1), f_i) = \pi(i)$. A directed heap path is π **connecting** l_1 and l_2, if $\pi(1) = l_1$ and $\pi(|\pi|) = l_2$, or vice versa (i.e., if $\pi(1) = l_2$ and $\pi(|\pi|) = l_1$).*

*A heap path π **traverses through** l if there exists i, $1 \leq i \leq |\pi|$ such that $l = \pi(i)$.*

Definition D.2 (Reachability) *A location l_2 is **reachable from** a location l_1 in a memory state $\sigma = \langle \rho, L, h, t, m \rangle$, if there is a directed heap path in h going from l_1 to l_2.*

Definition D.3 (Reachable locations) *A locations l is **reachable** in σ if it is reachable from a location which is pointed to by some variable. We denote the set of **reachable locations** in $\sigma \in \Sigma$ by $\mathcal{R}(\sigma)$, i.e., $\mathcal{R}(\sigma) = \{l \in L | x \in \mathcal{V} \text{ and } l \text{ is reachable in } \sigma \text{ from } \rho(x) \in Loc\}$.*

Definition D.4 (Connectivity) *Locations l_1 and l_2 are **connected** in a memory state $\sigma = \langle \rho, L, h, t, m \rangle$, if there is an* undirected *path in h connecting l_1 to l_2.*

Definition D.5 (Domination) *A set of locations $D \subseteq Loc$ are **dominator** (or **dominate their reachable heap**) in memory state $\sigma = \langle \rho, L, h, t, m \rangle \in \Sigma$, if for every $x \in \mathcal{V}$ such that $l_x = \rho(x) \in Loc$, every directed heap path in h* from l_x to *a location which is reachable from a location in D, traverses through a location in D.*

D.2 Components

We formalize the notion of components, implicit components decomposition using the definitions of *reachability* given in App. D.1. We use the auxiliary function $succ$, defined as $succ_h(L) = \{h(l)f \in Loc \mid l \in L, f \in \mathcal{F}\}$. $succ_h(L)$ is the set of objects which, in heap h, are pointed to by a field of an object in L.

Definition D.6 (Components) *The* domain of components *is* $\mathcal{C} = 2^{Loc} \times 2^{Loc} \times 2^{Loc} \times \mathcal{H} \times \mathcal{TM} \times \mathcal{M}$. *A* component $c = \langle I, L, R, h, t, m \rangle \in \mathcal{C}$ *is a* 6*-tuple. L contains c's internal objects; $I \subseteq L$ and $R \subseteq Loc \backslash L$ constitute c's spatial interface. The heap $h \in L \hookrightarrow \mathcal{F} \hookrightarrow (L \cup R \cup \{null, \ominus\})$*

defines the values of fields for objects inside c. The type map $t \in (L \cup R) \rightarrow \mathcal{T}$ *defines the types of the objects inside c and in its rim.* m *is c's* component module. *We say that component c belongs to* m*. For every* $l \in L, m(t(l)) = m$*. For every* $l \in R, m(t(l)) \neq m$*.*

A component can be in one of two states: sealed *or* unsealed*. In the context of a memory state, we are always able to tell the state of a component (see below). Thus, formally, we only place an additional restriction on sealed components: If c is a sealed component then* $R \subseteq \{h(l)f \mid l \in L, f \in \mathcal{F}\}$*.*

The types of the reachable objects in a memory state σ induce a (unique) *implicit component decomposition* of σ: The current component contains all the reachable locations that belong to the current module. Every sealed component contains a maximal set of reachable $\mathcal{M}$-connected locations. (These sets are mutually disjoint by definition).

Definition D.7 ($\mathcal{M}$-Connectivity) *Locations* l_1 *and* l_2 *are* ***$\mathcal{M}$-connected*** *in a memory state* $\sigma = \langle \rho, L, h, t, m \rangle$*, denoted by* $l_1 \overset{\mathcal{M}}{\leftrightsquigarrow}_\sigma l_2$*, if both of them belong to the same module in* σ *and there is an* undirected *path in* h *connecting* l_1 *to* l_2*. I.e.,*
$l_1 \overset{\mathcal{M}}{\leftrightsquigarrow}_\sigma l_2$ *iff there exists a sequence* $\pi \in \{1, \ldots, n\} \mapsto L$ *for some* $n \in \mathcal{N}$ *such that:*
(i) $l_1 = \pi(1),\ l_2 = \pi(n)$,
(ii) $m_\sigma(l_i) = m_c$ *for* $1 \leq i \leq n$ *and* $f_i \in \mathcal{F}$*, and*
(iii) $h(\pi(i), f_i) = \pi(i+1) \vee h(\pi(i+1), f_i) = \pi(i)$ *for* $1 \leq i \leq n-1$ *and* $f_i \in \mathcal{F}$*.*

The internal structure of an implicit component is induced by a *restriction* of a $\mathcal{DOS}$ memory state σ on a set of *reachable* locations who belong to the same module. Note that references from unreachable locations do not count. Also note that because our semantics is cutpoint-free, *unreachable locations in the local heap are also unreachable in the global heap* [39].

Definition D.8 (Implicit components) *A sealed component* $c \in \mathcal{C}$ *is an* ***implicit sealed component*** *of a* $\mathcal{DOS}$ *memory state* $\sigma = \langle \rho, L, h, t, m \rangle$ *if there exists a module* $m^c \neq m$ *and a set* $L^c \subseteq \mathcal{R}(\sigma)$ *of reachable objects such that for every* $l_1, l_2 \in L^c, l_1 \overset{\mathcal{M}}{\leftrightsquigarrow} \sigma l_2$ *such that* $c = \langle I, L^c, R, h|_{L^c}, t|_{L^c}, m^c \rangle$ *where* $I = \{l \in L^c \mid l \in succ_h(\mathcal{R}(\sigma))\} \cup \{\rho(x) \in L^c \mid x \in \mathcal{V}\}$ *and* $R = succ_h(L^c) \setminus L^c$*. (Note that, in particular,* $m_\sigma(l_1) = m_\sigma(l_2) = m_c$*.)*

The ***implicit current component*** *of* σ *is an unsealed component* $c \in \mathcal{C}$ *such that* $c = \langle I, L^*, R, h|_{L^*}, t|_{L^*}, m \rangle$ *where* $L^* = \{l \in \mathcal{R}(\sigma) \mid m_\sigma(l) = m\}$*,* $I = \{\rho(x) \in L^* \mid x \in \mathcal{V}\}$*, and* $R = (succ_h(L^*) \cup \{\rho(x) \in Loc \mid x \in \mathcal{V}\}) \setminus L^*$*.*

We denote the ***set of sealed components in a memory state*** $\sigma \in \Sigma$ *by* $\mathcal{C}(\sigma)$*. The implicit current component of* σ *is denoted by* $c^\star(\sigma)$*.*

Note that the entry locations of the unsealed current component are determined only by the values of variables. The acyclicity of the module dependency relation ensures that (in the local heap) there are no references from objects inside sealed components to objects inside the current component.

The component decomposition of a memory state σ naturally induces its *implicit component graph*, which is a directed graph whose nodes are the implicit components of σ and its edges reflect the inter-component reference structure.

$\langle \mathtt{x} = \mathtt{null}, \sigma\rangle \overset{D}{\leadsto} \langle \rho[x \mapsto \mathit{null}], L, h, t, m\rangle$	
$\langle \mathtt{x} = \mathtt{y}, \sigma\rangle \overset{D}{\leadsto} \langle \rho[x \mapsto \rho(y)], L, h, t, m\rangle$	$\textsc{acc}\;\; \rho(y) \neq \ominus$
$\langle \mathtt{x} = \mathtt{y.f}, \sigma\rangle \overset{D}{\leadsto} \langle \rho[x \mapsto h(\rho(y), f)], L, h, t, m\rangle$	$\textsc{cur}\;\; m_\sigma(\rho(y)) = m$
$\langle \mathtt{y.f} = \mathtt{x}, \sigma\rangle \overset{D}{\leadsto} \langle \rho, L, h[(\rho(y), f) \mapsto \rho(x)], t, m\rangle$	$\textsc{cur}\;\; m_\sigma(\rho(y)) = m$
	$\textsc{acc}\;\; \rho(x) \neq \ominus$
$\langle \mathtt{x} = \mathtt{alloc\ T}, \sigma\rangle \overset{D}{\leadsto}$	$\textsc{new}\;\; l \in \mathit{Loc} \setminus L$
$\quad \langle \rho[x \mapsto l], L \cup \{l\}, h[l \mapsto I], t[l \mapsto T], m\rangle$	$\textsc{typ}\;\; m(T) = m$
$\langle \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}), \sigma\rangle \overset{D}{\leadsto} \sigma$	$\textsc{cmp}\;\; \rho(x) \bowtie \rho(y)$
	$\textsc{acc}\;\; \rho(x) \neq \ominus,\; \rho(y) \neq \ominus$

Figure 9: Axioms for intraprocedural statements. $\sigma = \langle \rho, L, h, t, m\rangle$. The I function nullifies the fields of a newly allocated location. $\bowtie$ stands for either $=$ or $\neq$. When convenient, we sometimes treat h as an uncurried function, i.e., as a function from $\mathit{Loc} \times \mathcal{F}$ to Val.

Definition D.9 (Induced component graphs) *The* induced component graph *of a memory state σ denoted by $\mathcal{CG}(\sigma)$, is a directed graph $\mathcal{CG}(\sigma) = \langle \mathcal{C}(\sigma), E\rangle$ such that $E \subseteq \mathcal{C}(\sigma) \times \mathcal{C}(\sigma)$ and $\langle c_1, c_2\rangle \in E$ iff $R_1 \cap I_2 \neq \emptyset$, where $c_1 = \langle I_1, L_1, R_1, h_1, t_1, m_1\rangle$ and $c_2 = \langle I_2, L_2, R_2, h_2, t_2, m_2\rangle$.*

A component graph is ensured to be connected: · the target of a reference between two objects located inside different implicit components is also in the rim of the component containing the reference's source. · References from a local variable to a location outside the current module are treated as an inter-component reference leaving the current-component. In particular, all such entry locations are also in the rim of the implicit current component.

D.3 Operational Semantics

The meaning of basic statements and $Call$ and Ret operations is described by a transition relation $\overset{D}{\leadsto} \subseteq (\Sigma \times \mathit{stms}) \times \Sigma$. The semantics for interprocedural statements is given in Sec. 3.2. We now define the semantics for intraprocedural semantics.

Fig. 9 defines the axioms for intraprocedural statements. These are handled as in a 2-level store semantics for pointer programs. (`assume` statements are used to implement conditionals). The main difference is in the rules, expressed as side-conditions, regarding reference access (in particular, in the manipulation of pointer fields.) In short, these rules ensure that: - only fields of objects in the current component can be manipulated ($\textsc{cur}$); - only types of the current module can be instantiated ($\textsc{typ}$); and - inaccessible values are not accesed ($\textsc{acc}$).

D.4 Properties of the $\mathcal{DOS}$ Semantics

In this section, we formally define the notions of *observational soundness* and of *simulation* between the $\mathcal{DOS}$ semantics and the standard semantics. To be precise, when referring to the standard semantics we refer to the standard store-based semantics for pointer programs as defined by the $\mathcal{GSB}$ semantics in App. B.5.

Access paths We introduce access paths, which are the only means by which a program can observe a state. Note that the program cannot observe location names.

Definition D.10 (Field Paths) *A **field path** $\delta \in \Delta = \mathcal{F}^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by ϵ.*

Definition D.11 (Access path) *An **access path** $\alpha = \langle x, \delta \rangle \in AccPath = \mathcal{V} \times \Delta$ is a pair consisting of a local variable and a field path.*

Definition D.12 (Access path value in the $\mathcal{DOS}$ semantics) *The value of an access path $\alpha = \langle x, \delta \rangle$ in state $\sigma = \langle \rho, L, h, t, m \rangle$ of the $\mathcal{DOS}$ semantics, denoted by $[\![\alpha]\!]_{\mathcal{DOS}}(\sigma)$, is defined to be $\hat{h}(\rho(x), \delta)$, where*

$$\hat{h} \colon \mathit{Val} \times \Delta \to \mathit{Val} \text{ such that}$$
$$\hat{h}(v, \delta) = \begin{cases} v & \text{if } \delta = \epsilon \quad \text{(note that } v \text{ might be } \ominus \text{)} \\ \hat{h}(h(v, f), \delta') & \text{if } \delta = f\delta',\ v \in \mathit{Loc} \\ \bot & \text{otherwise} \quad \text{(note that } v \text{ might be } \ominus \text{)} \end{cases}$$

Note that traversal of the inaccessible value is not defined.

Definition D.13 (Comparable values) *A pair of values of the $\mathcal{DOS}$ semantics $v_1, v_2 \in \mathit{Val}$ are **comparable**, denoted by $v_1 \overset{?}{\bowtie} v_2$, $v_1 \neq \ominus$ and $v_2 \neq \ominus$.*

Definition D.14 (Access path value in the $\mathcal{GSB}$ semantics) *The value of an access path $\alpha = \langle x, \delta \rangle$ in state $\sigma_G = \langle \rho, L, h, t \rangle$ of the $\mathcal{GSB}$ semantics, denoted by $[\![\alpha]\!]_{\mathcal{GSB}}(\sigma_G)$, is defined to be $\overline{h}(\rho(x), \delta)$, where*

$$\overline{h} \colon \mathit{Val}_G \times \Delta \to \mathit{Val}_G \text{ such that}$$
$$\overline{h}(v, \delta) = \begin{cases} v & \text{if } \delta = \epsilon \\ \overline{h}(h(v, f), \delta') & \text{if } \delta = f\delta',\ v \in \mathit{Loc} \\ \bot & \text{otherwise} \end{cases}$$

Observational soundness We define the notion of observational equivalence between a $\mathcal{DOS}$ memory state σ and a standard 2-level store σ_G as the preservations in σ_G of all equalities and inequalities which hold in σ. Note that the preservation in the other direction is not required. Also note that an equality resp. inequality between access paths holds in σ only when the two access paths have comparable values. For simplicity, we define $[\![\texttt{null}]\!]_{\mathcal{DOS}}(\sigma) = [\![\texttt{null}]\!]_{\mathcal{GSB}}(\sigma) = \mathit{null}$.

Definition D.15 (Observational soundness) *The memory state $\sigma \in \Sigma$ is **observationally sound** with respect to memory state $\sigma_G \in \Sigma_G$, denoted by $\sigma_G \leqq \sigma$, if for every $\alpha, \beta \in AccPath \cup \{\texttt{null}\}$ it holds that*

$$\text{if } [\![\alpha]\!]_{\mathcal{DOS}} \overset{?}{\bowtie} [\![\beta]\!]_{\mathcal{DOS}} \text{ then } [\![\alpha]\!]_{\mathcal{DOS}}(\sigma)) = [\![\beta]\!]_{\mathcal{DOS}}(\sigma) \Leftrightarrow [\![\alpha]\!]_{\mathcal{GSB}}(\sigma_G) = [\![\beta]\!]_{\mathcal{GSB}}(\sigma_G)$$

We define the notion of observational soundness between two $\mathcal{DOS}$ memory states (resp. two standard memory states) in a similar manner.

Simulation Before we define the notion of simulation we briefly review some execution traces accessing-functions (formally defined in App. B). Given an execution trace π, the initial resp. final memory state of an execution trace π, denoted by $in(\pi)$ resp. $out(\pi)$, is the current memory state in the first resp. last stack of program states. $\pi(i)$ returns the stack at the ith step of the execution and $|\pi(i)|$ returns its height. $path(\pi)$ is the sequence of program points which the execution traverses. *i.e.*, $path(\pi)(i)$ is the program point in the ith step of the execution. (We assume that every statement is labeled by a program point.)

The following lemma shows that $\mathcal{DOS}$ simulates the standard semantics.

Theorem D.16 (Simulation) *Let π_S be a trace of a program P according to the standard semantics. There exists a trace π_D of P according to the $\mathcal{DOS}$ semantics such that either*

(i) $path(\pi_D) = path(\pi_S)$ and $out(\pi_S) \leqq out(\pi_D)$ or

(ii) $path(\pi_D)$ is a prefix of $path(\pi_S)$ and π_D gets stuck.

Lemma D.17 *Let P be a dynamically encapsulated program. The following holds:*

[Invariants] *An invariant concerning equality of access paths in the $\mathcal{DOS}$ semantics is an invariant in the standard semantics*

[Cleanness] *P does not dereferences null-valued pointer in the standard semantics.*

Lemma D.18 *Let P be a dynamically encapsulated program. A reference, that at a given program point always has the inaccessible value, is not live at that program point in the standard semantics.*

Definition D.19 (Observational equivalence) *The $\mathcal{DOS}$ memory states $\sigma_1, \sigma_2 \in \Sigma$ are **observationally equivalent**, denoted by $\sigma_1 \lesseqgtr \sigma_2$, if $\sigma_1 \leqq \sigma_2$ and $\sigma_2 \leqq \sigma_1$.*

The following lemma shows that $\mathcal{DOS}$ is indifferent to location names.

Theorem D.20 (Indifference to location names) *Let π_1, π_2 be execution traces of a program P according to the $\mathcal{DOS}$ semantics. If $|\pi_1(1)| = |\pi_2(1)| = 1$, $in(\pi_1) \lesseqgtr in(\pi_2)$ and $path(\pi_1) = path(\pi_2)$ then $out(\pi_1) \lesseqgtr out(\pi_2)$.*

E Module Invariants

Module invariants of a module m are properties pertaining to components of module m that are true in *any* program that uses m. We distinguish between two types of module invariants: *external module invariants* and *internal module invariants* pertaining to sealed and current components of module m, respectively.

Definition E.1 (Module invariants) *The* ***module invariant of type*** T ***of module*** $\mathbf{m}$*, denoted by* $[\![Inv_m\, T]\!] \subseteq 2^{\mathcal{C}}$*, is the set containing every sealed component of module* m *whose header is of type* T *in any memory state that may arise during an execution of any program* P *that uses* m*.*

$$[\![Inv_m\, T]\!] = \left\{ c \;\middle|\; P \textit{ ues } m, \sigma = \langle \rho, L, h, t, m \rangle \in \mathcal{R}(pp)^P, c \in \mathcal{C}(\sigma), t(hdr(c)) = T \;\right\} .$$

We say that a set S is a *sound external module invariant of module* $\mathbf{m}$ *for a type* T *of module* m if $[\![Inv_m\, T]\!] \subseteq S$.

Definition E.2 (Module implementation invariant) *The* ***Module implementation invariant at program point*** pp ***in procedure*** p ***of module*** $\mathbf{m}$*, denoted by* $[\![Inv_m^{imp}\; pp]\!] \subseteq 2^{(\mathcal{E}\times\mathcal{C})\times(\mathcal{E}\times\mathcal{C})}$*, is the set of pairs of an environment and a component such that* $\langle\langle \rho_e, c_e^\star\rangle, \langle \rho_x, c^{\star\prime}\rangle\rangle \in [\![Inv_m^{imp}\; pp]\!]$ *iff there exists a program* P *which uses* m *and two* $\mathcal{DOS}$ *memory states* σ_e *and* σ' *such that*
(i) $\sigma_e = \langle \rho_e, L_e, h_e, t_e, m\rangle$ *and* $c^\star(\sigma_e) = c_e^\star$*;*
(ii) $\sigma' = \langle \rho', L', h', t', m\rangle$ *and* $c^\star(\sigma') = c^{\star\prime}$*; and*
(iii) $\langle \sigma_e, \sigma'\rangle \in [\![pp]\!]^P$*.*

$$[\![Inv_m^{imp}\; pp]\!] = \bigcup_{P \textit{ calls } p} \left\{ \langle\langle \rho_e, c^\star(\sigma_e)\rangle, \langle \rho', c^\star(\sigma')\rangle\rangle \;\middle|\; \begin{array}{c} \sigma_e = \langle \rho_e, L_e, h_e, t_e, m\rangle, \\ \sigma' = \langle \rho', L', h', t', m\rangle, \\ \langle \sigma_e, \sigma'\rangle \in [\![pp]\!]^P \end{array} \right\} .$$

Note that by Inv. 1 (i), a local variable in an (dynamically encapsulated) memory state can point only to an object which is inside the current component or in its rim.

A set S is a *sound internal module invariant at program point* pp *in procedure* p *of module* m if $[\![Inv_m^{imp}\; pp]\!] \subseteq S$.

We define *the program independent meaning of a procedure*, denoted by $[\![Inv_m^{imp}\; p]\!]$, to be the module implementation invariant at the exit-site of p.

Definition E.3 (Program independent meaning of a procedure) *The* ***program independent meaning of a procedure*** p ***of module*** m*, denoted by* $[\![Inv_m^{imp}\; p]\!]$*, is* $[\![Inv_m^{imp}\; p]\!] = [\![Inv_m^{imp}\; e_p]\!]$*, where* e_p *is the exit point of procedure* p*.*

$[\![Inv_m^{imp}\; p]\!]$, is the module implementation invariant at the exit-site of p. A set S is a *sound modular meaning of a procedure* p if $[\![Inv_m^{imp}\; p]\!] \subseteq S$.

F Trimming Semantics

In this section, we define the *trimming semantics* and show how it approximates the $\mathcal{DOS}$ semantics. We describe certain properties of the trimming semantics which are of importance to our modular analysis.

To simplify notation, we assume I with (optionally) a certain index (resp. prime) to be the entry-points element of a component c with the same index (resp. prime). We use the same convention for (optionally) indexed (or primed) versions of L, R, h, t, and m.

F.1 Trimmed Memory States

The trimming semantics manipulates *trimmed memory states*. Intuitively, a trimmed state approximates a $\mathcal{DOS}$ memory state by abstracting a way all the information regarding sealed components and the structure of the component tree. Furthermore, the trimming semantics identifies *isomorphic* trimmed states.

Definition F.1 (Trimmed memory states) *The **domain of trimmed states** is $\sigma^\star, \langle \rho, c^\star \rangle \in \Sigma^\star = \mathcal{E} \times \mathcal{C}$. A **trimmed state** $\sigma^\star = \langle \rho, c^\star \rangle = \langle \rho, \langle I, L, R, h, t, m \rangle \rangle \in \Sigma^\star$ is a pair of an environment and an unsealed component. The only locations that the environment may map variables to are the entry-locations of $c^\star$ and the locations of its rim-objects, i.e., $range(\rho) \subseteq I \cup R \cup \{null, \ominus\}$.*

To define the notion of trimmed-state isomorphism, we first define components-isomorphism. Intuitively, two components are isomorphic if they are of the same module and one components can be produced from the other one by consistently renaming the locations of the other component.

Definition F.2 (Component isomorphism) *Component c_1 and $c_2 = \langle I_2, L_2, R_2, h_2, t_2, m_2 \rangle$ are **isomorphic according to a bijective function** $i : Loc \to Loc$, denoted by $c \sim_i c'$, if $c_1 = \langle i \circ I_2, i \circ L_2, i \circ R_2, sub_i \circ h_2 \circ i^{-1}, m_2 \rangle$, where i is extended pointwise to sets of locations. Components c_1 and c_2 are **isomorphic**, denoted by $c_1 \sim c_2$, if there exists a bijective function $i : Loc \to Loc$ such that $c_1 \sim_i c_2$.*

Definition F.3 (Trimmed-states isomorphism) *Trimmed-states $\sigma_1^\star = \langle \rho_1, c_1^\star \rangle$ and $\sigma_2^\star = \langle \rho_2, c_2^\star \rangle$ are **isomorphic according to an bijective function** $i : Loc \to Loc$, denoted by $\sigma_1^\star \sim_i \sigma_2^\star$, if $c_1 \sim_i c_2$ and $\rho_1 = sub_i \circ \rho_2$. Trimmed-states $\sigma_1^\star$ and $\sigma_2^\star$ are **isomorphic**, denoted by $\sigma_1^\star \sim \sigma_2^\star$, if there exists a bijective function $i : Loc \to Loc$ such that $\sigma_1^\star \sim_i \sigma_2^\star$.*

We extend the isomorphism relation $\sim$ in a pointwise manner to sets of trimmed memory sets. We say that $S_1, S_2 \subseteq \Sigma^\star$ are *isomorphic*, denoted by $S_1 \sim S_2$, when for every $\sigma_1 \in S_1$ there exists $\sigma_2 \in S_2$ such that $\sigma_1 \sim \sigma_2$, and vice versa.

We refer to a bijective total function $i : Loc \to Loc$ as a location renaming function. Note that the same location renaming function, i, is applied to the environment, ρ_2, and to the location sets in in component, c_2, of $\sigma_2^\star$. This ensures a consistent change of the values of pointer variables with the location renaming.

We define the Galois connection between the power-domain of memory states of the $\mathcal{DOS}$ semantics and the power-domain of trimmed states in a pointwise manner. We define the abstraction of single $\mathcal{DOS}$ memory states by: (i) defining a function which maps every $\mathcal{DOS}$ memory state σ to a trimmed memory state $\sigma^\star$, and (ii) considering every trimmed states which is isomorphic to $\sigma^\star$ as an abstraction of σ.

Definition F.4 (Abstraction) *The **trimmed memory state induced by a $\mathcal{DOS}$ memory state** σ, denoted by $trim(\sigma)$, is $\langle \rho, c^\star(\sigma)\rangle$. A trimmed memory state $\sigma^\star$ is an **abstraction** of a $\mathcal{DOS}$ memory state σ, denoted by $\sigma \sqsubseteq \sigma^\star$, if $\sigma^\star \sim trim(\sigma)$.*

The trimmed state abstraction induces a natural equivalence relation between $\mathcal{DOS}$ memory states. We say that $\sigma_1, \sigma_2 \in \Sigma$ are *equivalent under the trimmed abstraction*, denoted by $\sigma_1 \sim_\star \sigma_2$ when they are abstracted by the same set of trimmed memory states, i.e., when $trim(\sigma_1) \sim trim(\sigma_2)$.

We extend $trim$ pointwise to sets of $\mathcal{DOS}$ memory states, *i.e.*, for a set $S \subseteq \Sigma$ of $\mathcal{DOS}$ memory states, $trim(S) = \{trim(\sigma) \mid \sigma \in S\}$. We also extend the $\sim_\star$ to sets of $\mathcal{DOS}$ memory states. We say that $S_1, S_2 \subseteq \Sigma$ are *equivalent under the trimmed abstraction*, denoted by $S_1 \sim_\star S_2$, if $trim(S_1) \sim trim(S_2)$.

F.2 Induced Operational Semantics

In this section, we define the operational semantics of the trimming semantics using the best transformer [12]. We show that the meaning of all the statements, except the ones pertaining to the $Call$ operations of intermodule procedure invocations, are complete [15]. Thus, it is possible to determine the effect of a statement on a trimmed memory state $\sigma^\star$ by applying it to any (admissible) $\mathcal{DOS}$ memory states σ abstracted by $\sigma^\star$. Furthermore, we show how to determine a set of such $\mathcal{DOS}$ memory states for every trimmed state.

Intuitively, the reason why the best transformers pertaining to all statements except for the aforementioned $Call$ operations are complete is that these statements do not require information regarding the content of sealed components. The $Call$ operations pertaining to intermodule procedure calls, on the other hand, do require such information on the contents of the components pointed to by the actual parameters. This information, however, is not maintained in the trimmed state.

F.2.1 Minimal $\mathcal{DOS}$ States

We construct a set of $\mathcal{DOS}$ memory states which are abstracted by a given trimmed memory state $\sigma^\star = \langle \rho, c^\star \rangle$ by, basically, constructing a $\mathcal{DOS}$ memory state σ in which all the locations inside $c^\star$ are in σ's current component and every location in the rim of $c^\star$ is placed in its own subcomponent of σ's (constructed) current component.

Definition F.5 (Minimal $\mathcal{DOS}$ states) *The **minimal $\mathcal{DOS}$ memory state corresponding to a trimmed memory state** $\sigma^\star = \langle \rho, \langle I, L, R, h, t, m\rangle\rangle \in \Sigma^\star$, denoted by $dos(\sigma^\star)$, is $\sigma = \langle \rho, L \cup R, h, t, m\rangle$.*

Lemma F.6 *Let $\sigma \in \Sigma$ be an admissible $\mathcal{DOS}$ memory state. Let $\sigma^\star \in \Sigma^\star$ be a trimmed memory state such that $\sigma^\star \sim trim(\sigma)$. Then (i) $dos(\sigma^\star)$ is dynamically encapsulated and (ii) $\sigma^\star \sim trim(dos(\sigma^\star))$.*

F.2.2 Intraprocedural Statements

We define the meaning of intraprocedural statements in the trimming semantics by, essentially, executing them using the $\mathcal{DOS}$ semantics on minimal $\mathcal{DOS}$ states.

Definition F.7 (Intraprocedural trimming semantics) *The **trimming semantics of an intraprocedural statement** $st \in Stmt$ is a relation $[\![st]\!]_\star \subseteq \Sigma^\star \times \Sigma^\star$ where $\sigma_1^\star \in [\![st]\!]_\star(\sigma_2^\star)$ iff there exist $\sigma_1 \sqsubseteq \sigma_1^\star$ and $\sigma_2 \sqsubseteq \sigma_2^\star$ such that $\sigma_1 \in [\![st]\!]_{\mathcal{DOS}}(\sigma_2)$.*

Lemma F.8 (Equivalence preservation) *Let $st \in Stmt$ be an intraprocedural statement. Let $\sigma_1, \sigma_2 \in \Sigma$ be $\mathcal{DOS}$ memory states. If $\sigma_1 \sim_\star \sigma_2$ then $[\![st]\!](\sigma_1) \sim_\star [\![st]\!](\sigma_2)$.*

Sketch of Proof: Follows directly from The. D.20.

Corollary F.9 (Completeness of intraprocedural trimming semantics) *Let $st \in Stmt$ be an intraprocedural statement. If $\sigma \sqsubseteq \sigma^\star$ then $[\![st]\!]_\star(\sigma^\star) \sim_\star trim([\![st]\!]_{\mathcal{DOS}}(\sigma))$.*

F.2.3 Interprocedural Statements

In the rest of this section, we assume that $y = p(x_1, \ldots, x_k)$ is an arbitrary procedure invocation statement, and that p's formal variables are $z_1, \ldots, z_k$. We also use the following shorthands $\sigma_c = \langle \rho_c, L_c, h_c, t_c, m_c \rangle$, $\sigma_e = \langle \rho_e, L_e, h_e, t_e, m_e \rangle$, $\sigma_x = \langle \rho_x, L_x, h_x, t_x, m_x \rangle$, and $\sigma_r = \langle \rho_r, L_r, h_r, t_r, m_r \rangle$.

F.2.3.1 Call Operation

We define the trimming semantics of a $Call$ operation in the same style we used to define the trimming semantics of intraprocedural statements. Note however that in case of intermodule procedure call, the only information known about the subcomponents pointed to by the actual parameters is the the type of their headers. Thus, the trimming semantics has to consider every possible component with an header of the appropriate type.

Definition F.10 (Interprocedural trimming semantics of $Call$ operations) *The **interprocedural trimming semantics of a $Call$ operation pertaining to a procedure invocation** $\mathbf{y} = \mathbf{p}(x_1, \ldots, x_k)$ is a relation $[\![Call_{y=p(x_1,\ldots,x_k)}]\!]_\star \subseteq \Sigma^\star \times \Sigma^\star$ where $\sigma_e^\star \in [\![Call_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma_c^\star)$ iff there exist $\mathcal{DOS}$ memory states $\sigma_1 \sqsubseteq \sigma_1^\star$ and $\sigma_2 \sqsubseteq \sigma_2^\star$ where $\sigma_1 \in [\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma_2)$.*

Lemma F.11 (Equivalence preservation) *Let $y = p(x_1, \ldots, x_k)$ be a procedure invocation statement. If $\sigma_1 \sim_\star \sigma_2$ then $[\![Call_{y=p(x_1,\ldots,x_k)}]\!]_{\mathcal{DOS}}(\sigma_1) \sim_\star [\![Call_{y=p(x_1,\ldots,x_k)}]\!]_{\mathcal{DOS}}(\sigma_2)$.*

Sketch of Proof: Both $\mathcal{DOS}$ memory states either simultaneously satisfy $Call$'s side-conditions or not. The resulting entry memory states are all possible $\mathcal{DOS}$ in which the formal parameter have the same values as their corresponding actual parameters at the call memory state.

Corollary F.12 (Completeness of intra-module $Call$ operations) *Let $y = p(x_1, \ldots, x_k)$ be a procedure invocation statement. If $m(\sigma) = m(p)$ and $\sigma \sqsubseteq \sigma^\star$ then* $[\![Call_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma^\star) \sim_\star trim([\![Call_{y=p(x_1,\ldots,x_k)}]\!]_{\mathcal{DOS}}(\sigma))$.

Note that Cor. F.12 applies only to intra-module procedure calls. It is clear that an execution of an intermodule procedure call on a given $\mathcal{DOS}$ memory state σ results in a specific memory state comprised of a combination of some of σ's current component's subcomponents. As indicated above, the trimming semantics abstracts away this information.

F.2.3.2 Return Operation Ret is a binary operation. Although the semantics does not distinguish between isomorphic trimmed memory states, our choice of using a store-based representation for the memory states instead of a storeless one (see, e.g., [36]) causes that certain combinations of $\mathcal{DOS}$ memory states which are abstracted by a pair of trimmed memory states cannot represent any possible call- and return- memory states, while other may. Thus, when we we use $\mathcal{DOS}$'s meaning for Ret for defining its meaning in the trimming semantics we would like not to consider pairs of call- and exit- memory states that may never occur in any program:

Definition F.13 *A $\mathcal{DOS}$ memory state σ_c is* ***possible call memory state for an inter-module procedure invocation*** *$y = p(x_1, \ldots, x_k)$ if:*
(i) $m(p) \neq m_c$,
(ii) for every i, $1 \leq i \leq k$, $\rho_c(x_i) \in Loc$,
(iii) for every i, j, $1 \leq i < j \leq k$, $\rho_c(x_i) \neq \rho_c(x_j)$, and
(iv) for every i, $1 \leq i \leq k$, m_c depends on $m(t(\rho_c(x_i))) \neq m_c$.

Definition F.14 *Two $\mathcal{DOS}$ memory states σ_c and σ_x are* ***possible call- and exit- memory states for a procedure invocation*** *$y = p(x_1, \ldots, x_k)$ if:*
(i) $L_c \cap L_x \subseteq \{\rho_c(x_i) \in Loc \mid 1 \leq i \leq k\}$,
(ii) for every $i = 1, \ldots, k$, if $\rho_x(z_i) \neq \ominus$ then $\rho_x(z_i) = \rho_c(x_i)$, and
(iii) if the invocation is an intermodule procedure call, i.e., $m(p) \neq m_c$, then σ_c is a possible call memory state for $y = p(x_1, \ldots, x_k)$.

Any pair of call- and exit- memory states that might result due to a procedure invocation in $\mathcal{DOS}$ satisfies (ii) because $\mathcal{DOS}$ is a cutpoint-free, local-heap semantics which never reuses allocated locations. It satisfies (iii) because formal variables are not assigned (but might be blocked).

Definition F.15 (Interprocedural trimming semantics of Ret operations) *The* ***interprocedural trimming semantics of a Ret operation pertaining to a procedure invocation*** $\mathbf{y} = \mathbf{p}(x_1, \ldots, x_k)$ *is a relation* $[\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star \subseteq (\Sigma^\star \times \Sigma^\star) \times \Sigma^\star$ *where $\sigma_r^\star \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma_c^\star, \sigma_x^\star)$ iff there exist $\mathcal{DOS}$ memory states $\sigma_c \sqsubseteq \sigma_c^\star$, $\sigma_x \sqsubseteq \sigma_r^\star$, and $\sigma_r \sqsubseteq \sigma_r^\star$, where σ_c and σ_x are possible call- and exit- memory states for a procedure invocation $y = p(x_1, \ldots, x_k)$ and $\sigma_r \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\sigma_c, \sigma_x)$.*

Lemma F.16 (Equivalence preservation) *Let $y = p(x_1, \ldots, x_k)$ be a procedure invocation statement. Let $\sigma_c^1 \sim \sigma_c^2$, $\sigma_x^1 \sim \sigma_x^2$ be $\mathcal{DOS}$ memory states. If σ_c^i and σ_x^i, for $i = 1, 2$, are possible call- and exit- memory states for a procedure invocation $y = p(x_1, \ldots, x_k)$, then $[\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\sigma_c^1, \sigma_x^1) \sim [\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\sigma_c^2, \sigma_x^2)$.*

Corollary F.17 (Completeness of Ret operations) *Let $y = p(x_1, \ldots, x_k)$ be a procedure invocation statement. $\sigma_c \sqsubseteq \sigma_c^\star$, $\sigma_x \sqsubseteq \sigma_x^\star$, and σ_c and σ_x are possible call- and exit- memory states for a procedure invocation $y = p(x_1, \ldots, x_k)$, then $[\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma_c^\star, \sigma_x^\star) \sim_\star trim([\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_{\mathcal{DOS}}(\sigma_c, \sigma_r))$.*

Lem. F.20 shows that a stronger property than Cor. F.17 holds for intermodule procedure calls.

Observation F.18 *To determine the effect of an intra-module statement on trimmed memory state $\sigma^\star$ it suffice to execute the statement on a minimal $\mathcal{DOS}$ memory state corresponding to $\sigma^\star$.*

Similarly to Obs. F.18, the effect of a Ret operation on two trimmed-memory states can be determined by applying Ret on the corresponding two minimal $\mathcal{DOS}$ memory states.

F.3 Properties of the trimming semantics

In this section, we establish two properties of the trimming semantics which enables to modularly characterize module invariants of the $\mathcal{DOS}$ semantics using the trimming semantics. These properties concern the handling of $Call$ and Ret operations pertaining to intermodule procedure calls. We start by discussing the Ret operation, which is simpler.

F.3.1 Intermodule Ret operations.

The Ret operation in the trimming semantics requires to construct (minimal) $\mathcal{DOS}$ memory states of different modules. In this section, we show how mitigate this requirement. Using the ownership transfer specification, we consider only (minimal) $\mathcal{DOS}$ call memory states and a very simple $\mathcal{DOS}$ memory state representing the exit-memory state. The latter can be *fabricated* (constructed) from the procedure specification.

Def. F.19 defines the notion of *shallowly similar* $\mathcal{DOS}$ memory states. This notion is (much) weaker than equivalence under isomorphism. Lem. F.20 utilizes this notion to show that the Ret operation has a very light dependency on the contents of the exit-state. Note that Lem. F.20 strengthen Lem. F.16.

Definition F.19 (Shallow similarity) *Two $\mathcal{DOS}$ memory states $\sigma_1, \sigma_2 \in \Sigma$ are **shallowly similar**, denoted by $\sigma_1 \overset{s}{\sim} \sigma_2$, if:*
(i) $m(\sigma_1) = m(\sigma_2)$,
(ii) $dom(\rho_1) = dom(\rho_2)$,
(iii) $\forall x \in dom(\rho_1) : \rho_1(x) = \ominus \iff \rho_2(x) = \ominus$,

(iv) $\forall x \in dom(\rho_1) : \rho_1(x) = \rho_1(ret) \iff \rho_2(x) = \rho_2(ret)$,
(v) $\rho_1(ret) \in Loc \iff \rho_2(ret) \in Loc$, *and*
(vi) $\rho_1(ret) = null \iff \rho_2(ret) = null$.

Note that if two $\mathcal{DOS}$ memory states are equivalent under abstraction, *i.e.*, $\sigma_1 \sim_\star \sigma_2$ implies $\sigma_1 \overset{s}{\sim} \sigma_2$.

Lemma F.20 (Strong equivalence preservation) *Let $\sigma_c^1 \sim \sigma_c^2$, $\sigma_x^1 \overset{s}{\sim} \sigma_x^2$ such that $m(\sigma_c^1) \neq m(\sigma_x^1)$. If σ_c^i and σ_x^i, for $i = 1, 2$ are possible call- and exit- memory states for an intermodule procedure invocation $y = p(x_1, \ldots, x_k)$, then $[\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\sigma_c^1, \sigma_x^1) \sim [\![Ret_{y=p(x_1,\ldots,x_k)}]\!](\sigma_c^2, \sigma_x^2)$.*

Procedure Specification We now utilize Lem. F.20 to harness procedure specification for determining the effect of inter-module procedure calls invoked by (the analyzed) module m.

Definition F.21 (Trimmed shallow memory state) *The* ***trimmed shallow memory state*** *corresponding to a trimmed memory state $\sigma^\star = \langle \rho, \langle I, L, R, h, t, m \rangle \rangle$, denoted by $shallow(\sigma^\star)$ is $\langle \rho, \langle I \cap E, L \cap E, R \cap E, \bot, t|_E, m \rangle \rangle$ where $E = \{\rho(x) \in Loc \mid x \in dom(\rho)\}$.*

Observation F.22 *For any $\sigma_1 \sqsubseteq \sigma_1^\star$ and $\sigma_2 \sqsubseteq \sigma_2^\star$, $\sigma_1 \overset{s}{\sim} \sigma_2$ iff $shallow(\sigma_1^\star) \sim shallow(\sigma_2^\star)$.*

Definition F.23 (Shallow procedure specification) *The* ***trimmed shallow procedure specification*** *for a procedure p of module m, with k formal parameters $z_1, \ldots, z_k$ denoted by $[\![p]\!]_\star^s$, is*

$$[\![p]\!]_\star^s = \left\{ \langle \sigma^\star, \sigma^{\star\prime} \rangle \;\middle|\; \begin{array}{c} \langle \sigma, \sigma' \rangle \in [\![Inv_m^{imp}\ p]\!], \\ \sigma^\star = shallow(trim(\sigma)), \\ \sigma^{\star\prime} = shallow(trim(\sigma')) \end{array} \right\} .$$

Any set $S \subseteq \Sigma^\star \times \Sigma^\star$ which is a superset of $[\![p]\!]_\star^s$ is a ***sound trimmed shallow specification of a procedure*** *p.*

We assume that we can fabricate arbitrary shallow memory states. Thus, given the specification, as defined in Sec. 2, of an arbitrary procedure p, we can fabricate a sound shallow specification of procedure p.

Theorem F.24 (Soundness of trimmed shallow specifications) *Let $\pi = \pi'\pi''\pi''' \in \Pi_{\mathcal{DOS}}^P$ be a feasible trace of a program P according to the $\mathcal{DOS}$ semantics such that $\pi'' \in \Pi_{\mathcal{DOS}}^{P,e_p}$ is a complete execution trace of some procedure p in program P.*
Let $y = p(x_1, \ldots, x_k)$ be an inter-module procedure call executed at the current program point in $\pi'(|\pi'|)$.
Let $\sigma_c = out(\pi')$ and $\sigma_r = in(\pi''')$ be the call- and return- memory states, correspondingly. Let $\sigma_c^\star = trim(\sigma_c)$ and $\sigma_r^\star = trim(\sigma_r)$.
Then, $\sigma_r^\star \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma_c^\star, [\![p]\!]_\star^s \circ [\![call_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma_c^\star))$.

Note that $[\![p]\!]^s_\star$ and $[\![call_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma^\star_c)$ can compose only when the latter produces trimmed shallow memory states. As a consequence, when executing an intermodule $Call$ operation in our analysis, we do not need to consider arbitrary contents for the sealed components. It is suffice to assume that each one contains a single object.

F.3.2 Intermodule $Call$ operations.

Lem. F.20 holds, because, in $\mathcal{DOS}$, the only effect of an intermodule procedure call on the environment and on the current component of the call memory state, σ_c, is that certain pointers variables and fields can become inaccessible. The variable being assigned the (pointer-valued) return value, ret, is assigned a location which, in the return memory state, is the header of a sealed component. This header may be one of the headers pointed to by an actual parameter, or a different one, which is not pointed to by any pointer variable or field in part of the heap of σ_c which is unavailable for the call.

Lem. F.31 shows that the the contents of the sealed component pointed to by the actual parameters in a $\mathcal{DOS}$ call memory state, σ_c, *determine* the trimming abstraction of the entry state, σ_e, constructed by a $Call$ operation pertaining to an inter-module procedure call.

We now define certain predicates and operations which operate on components and trimmed memory states. These operations allows us to explicitly fabricate memory state.

Definition F.25 (Disjointness of components) *Components c_1 and c_2 are **disjoint**, denoted by $c_1 \sharp c_2$, if $(L_1 \cup R_1) \cap (L_2 \cup R_2) = \emptyset$.*

Definition F.26 (Combination of components) *The **Combination** of disjoint components c_1 and c_2 denoted by $c_1 \oplus c_2$, is the component $\langle I_1 \cup I_2, L_1 \cup L_2, R_1 \cup R_2, h_1 \cup h_2, t_1 \cup t_2, m \rangle$.*

Definition F.27 (Empty component) *The **empty component of module** m, denoted by $c^m_\emptyset$, is $c^m_\emptyset = \langle \emptyset, \emptyset, \emptyset, \bot, \bot, m \rangle$,*

Definition F.28 (Rim component) *The **rim component** of a location l, a type T, and module m, such that $m(T) \neq m$, denoted by $c^m_{rim}(l,T)$, is $c^m_{rim}(l,T) = \langle \{l\}, \emptyset, \{l\}, \bot, [l \mapsto T], m \rangle$.*

We say that a component $c \in \mathcal{C}(\sigma)$ is the *component of a (reachable) location $l \in \mathcal{R}(\sigma)$ in memory state σ*, denoted by $c_\sigma(l)$, if l is inside c.

Definition F.29 ($\mathcal{M}$-projected component) *The **$\mathcal{M}$-projected component** of a variable x in a $\mathcal{DOS}$ memory state $\sigma = \langle \rho, L, h, t, m \rangle$ wrt. module m' such that $m_\sigma(x) \neq m$, denoted by $c^{m'}_\sigma(l)$, is*

$$c^{m'}_\sigma(x) = \begin{cases} c_\sigma(\rho(x)) & m(t(x)) = m' \text{ and } \rho(x) \in Loc \\ c^{m'}_{rim}(\rho(x), t(x)) & m(t(x)) \neq m' \text{ and } \rho(x) \in Loc \\ c^{m'}_\emptyset & \text{otherwise}\ . \end{cases}$$

The $\mathcal{M}$-projected component of a variable x in memory state $\sigma = \langle \rho, L, h, t, m\rangle$, in the conditions of, Def. F.29, is:

- the sealed component whose header, $\rho(x)$, is pointed to by x, if x points to a location of module $m' \neq m$;
- the rim component of the location pointed to by x, $\rho(x)$, and x's type, $t(\rho(x))$, if $t(\rho(x)) \neq m'$; or
- the empty component of module m', otherwise (i.e., if x does not point to a location).

Definition F.30 (Projection of Components) *The* ***projection*** *of component* $c = \langle I, L, R, h, t, m\rangle$ *on an entry-location* $l \in I$*, denoted by* $c|_l$*, is the component* $\langle I \cap L_{rel}, L \cap L_{rel}, R \cap L_{rel}, h|_{L_{rel}}, t|_{L_{rel}}, m\rangle$*, where* $L_{rel} = R_h(\{l\})$*.*

Memory state fabrication

Lemma F.31 (Inter-module entry state fabrication) *Let* $\sigma_c \in \Sigma$ *be a possible call memory state for a procedure invocation* $y = p(x_1, \ldots, x_k)$ *such that* $m(\sigma_c) \neq m(p)$*. Let* $c_i = c_{\sigma_c}^{m(p)}(\rho_c(x_i))$*, for* $i = 1, \ldots, k$*. Let* $\sigma_e^\star = \langle [z_i \mapsto \rho_c(x_i) \mid 1 \leq i \leq k], \sigma_1^\star \oplus \ \ldots \oplus \sigma_k^\star\rangle$*. Then,* $\sigma_e^\star = trim([\![Call_{y=p(x_1,\ldots,x_k)}]\!]_{\mathcal{DOS}}(\sigma_c))$*.*

Sketch of Proof: The proof is based on the following properties of the $\mathcal{DOS}$ semantics: every formal parameter in an entry memory state σ_e resulting from an inter-module procedure call dominates its reachable subheap. In addition, in σ_e, different formal parameters point to disjoint subheaps. Also procedures of module m can manipulate only memory states whose current component is of module m.

Thus, in any trimmed memory state $\sigma_e^\star$ such that $\sigma_e \sqsubseteq \sigma_e^\star$, a formal parameter whose type is not of module m, points to an isolated object whose fields are undefined. The current component is the only component of module m. It is implicitly created by reassigning to the current component of the entry memory state all the locations inside the subcomponents of the current component of the call memory state whose headers are by an actual parameter. In particular, the subheap comprising every such subcomponent is not mutated.

We utilize Lem. F.31 to conservatively determine every possible input state to a procedure. We assume that we can fabricate empty components and rim components of arbitrary modules. The challenge is to determine the possible sealed components of (the analyzed) module m. Lem. F.32 shows that in any program, such components were sealed when a preceding inter-module procedure call was invoked. Furthermore, no other module could have modified these components, as guaranteed by the program model (see Sec. 2).

Lemma F.32 *[Consistency of sealed components] Let* $\pi \in \Pi_{\mathcal{DOS}}^P$ *be a feasible trace of an arbitrary program* P *according to the* $\mathcal{DOS}$ *semantics. Let* $c \in \mathcal{C}(\sigma)$ *be a sealed component of module* m *in the* $\mathcal{DOS}$ *memory state* $\sigma = out(\pi)$*. Then, there exists a prefix* π' *of* π *such that*

(i) $\langle e_p, \sigma_x\rangle = top(\pi(|\pi'|))$ *is an exit program state of some procedure* p *of module* m*.*

(ii) $m(cur_{proc}(pop(\pi(|\pi'|)))) \neq m$*, the caller of* p *is not of module* m*.*

(iii) There exists a location l which in memory state σ_x is pointed to by a non-transferred formal variable or by the ret variable (the return value), $l \in \{\rho_x(ret), \rho_x(z_i) \mid 1 \le i \le k\} \cap Loc$ (assuming that p's formal variables are $z_1, \ldots, z_k$), such that $c_\sigma(l) \sim c^\star(\sigma_x)|_l$.

G Two Approaches for Modular Analysis

In this section, we present two approaches for modular analysis. Sec. G.1 describe the approach using the most-general-client of a module and provides a scheme for constructing one. Sec. G.2 describe the approach using fixpoint equation system derived from the code of the analyzed module and the specification of the modules it depends on.

G.1 Most General Client

Fig. 10 defines the most general client of an arbitrary module m_i, a program which non deterministically invokes every procedure of module m_i. The most challenging aspect of the most general client is computing every possible value for the actual parameters.

The value of an actual parameter x_i of type T_j of module m_i which is used in a procedure invocation of module m is obtained by invoking a procedure $\texttt{mgc\_m}_i\texttt{\_T}_j$, which returns all the possible sealed components (along with their subcomponent trees) whose header is of type T_i.

Procedure $\texttt{mgc\_m}_i\texttt{\_T}_j$ acts as the most general client of module m_i and returns an arbitrary component whose header is of type T_j. Procedure $\texttt{mgc\_m}_i\texttt{\_T}_j$ works by non deterministically invoking every procedure p_k in module m_i that either have a formal parameter of type T_j or which returns a value of that type. The invocation of p_k is done by calling procedure `mgc_m_i_T_j_p_k()` which returns a pointer to the required component. In case procedure p_k has several formal parameters of type T_j or both a formal parameter of type T_j and and a return variable of that type, the procedure returns one of them in a non deteministic fashion.

Note that becasue m_i might be m, the invoked $\texttt{mgc\_m}_i\texttt{\_T}_j$ procedure might return a component of module m. Thus, the most general client of a module m is simply a program that loops and non deterministically invokes the procedures $\texttt{mgc\_m\_T}_j$ for every type T_j of module m.

The most general client can produce every possible input parameters to a procedure p of module m that might arise in a feasible trace. Intuitively, the most-general-client can produce these inputs because there is no aliasing between parameters. In particular, even if the most-general client needs to handle a case where a procedure is invoked with 2 pointers parameters, say x_1 and x_2, that point to components that were supposed to be sealed by the same (previous) call to a procedure of the analyzed module, it can separately repeat the sequence of calls that led to that call for x_1 and x_2: First, it can compute the value of x_1, and then, the value of x_2.

```
mgc_m_i () {
 while (true) {
  T_1 t_1 = mgc_m_i_T_1() ||
  T_1 t_2 = mgc_m_i_T_2() ||
  ...
  T_k_i t_k_i = mgc_m_i_T_k_i();
 }
}
```

(a)

```
T_j mgc_m_i_T_j () {
 T_j t_j = null;
  while (true) {
   t_j = mgc_m_i_T_j_p_j1(); ||
   t_j = mgc_m_i_T_j_p_j2(); ||
   ...
   t_j = mgc_m_i_T_j_p_jh_j();||
   if (t_j != null) return t_j;
  }
 }
```

(b)

```
// We assume procedure p_jn has k formal parameters,
// z_1, ..., z_k. The type of parameter z_i is T_f_i and
// m_f_i = m(T_f_i). We assume that the variables
// x_jn_1,...,x_jn_m_j are the actual parameters
// of type T_j such that their corresponding formal
// parameters are in F^nt_p_jn. If the type of p_jn's
// return value, T_j0, is of type T_j then we assume
// that there is an x_j_l which is x_0.
T_j mgc_m_i_T_j_p_jn() {
 T_f1 x_1 = mgc_m_f1_T_f1();
 T_f2 x_2 = mgc_m_f2_T_f2();
 ...
 T_fk x_k = mgc_m_fk_T_fk();

 T_j0 x_0 = p_jn(x_1,...,x_k);

 T_j ret = null;
 ret = x_j1 || ...|| ret = x_jm;
 return ret;
}
```

(c)

Figure 10: Pseudo code of the most general client of a module m. `||` denotes non deterministic choice: `st1 || st2;` means that either statement `st1` is executed or `st2`. We assume there are k_i user defined types of module $\mathtt{m}_i$. We assume that the only procedures of module $\mathtt{m}_i$ that have either a formal parameter of type $\mathtt{T}_j$ or that their return value is of type $\mathtt{T}_j$ are procedures $\mathtt{p}_{j_1} \ldots \mathtt{p}_{j_{h_j}}$. (a) The most general client of module m_i. (b) A procedure that returns the header of an arbitrary sealed component of module m_i whose header is of type T_j. (c) A procedure that returns an arbitrary sealed component of module m_i whose header is of type T_j which can result at the return-site from procedure p_{j_n}.

Theorem G.1 *Let P be a program. Let π be an admissible program trace of P. Let l be the header of a sealed component in memory state $\sigma = cur_{state}(out(\pi))$ whose type is T_j of module m_i. There exists an execution trace of the most general client of module m_i such that variable $\mathtt{t}_j$ of procedure $\mathtt{mgc\_m}_i$ points to a sealed component which is isomorphic to that of component $c_\sigma(l)$.*

G.2 An Equation System Definition of Sound Module Invariants

In this section, we present an equation system whose (fixpoint) solution determines sound module invariants and module implementation invariants (Def. E.2 and Def. E.1, respectively) of module m. The equation system is specified based on the trimming semantics and the *bodies* of the procedures of module m (Sec. 2). The effect of intermodule procedure calls made by procedures of module m is determined using the invoked procedures *specification* (Sec. 2).

The equation system is, in most parts, a standard data-flow-like based equation system, aimed at determining the collecting semantics (relational trimming-semantics, in our case) of a program. The challenge is that we do not want to analyze the module in the context of a *specific* program but in the context of *any* program, nor do we want to analyze the procedures of modules used by m. Sec. F.3.2 describes how we achieve the above by *fabricating* every possible entry memory state to any procedure of module m in any program. Sec. F.3.1 describes how we utilize procedure specification to handle procedure calls. Fig. 11 presents the equation system.

We note that although the equation system is defined in terms of trimmed memory states of (*only*) module m. It does not lead to an *effective* modular analysis *algorithm*: In general, trimmed memory states might be of an unbounded size. Thus, an algorithmic solution might be out of hand. However, one can be derived, if the trimming semantics is replaced by a semantics which approximates it in a bounded way, e.g., using the abstraction presented in App. H.

Fig. 11 provides an equation system for which any (fixpoint) solution determines a sound module invariants of module m. The equation system simultaneously determines the module implementation invariants. For simplicity, and without loss of generality, we assume that the procedures of module m are either *interface procedures*, which may be invoked by procedures from modules other than m, and *private procedures* which may be invokes only be procedures from module m.

The equation system utilizes trimmed shallow procedure specifications and to handle inter-module procedure calls. It determines all possible entry memory states for every procedure of module m by combining, in every possible way, the sealed components that were already found.

Note that the determined invariants are sound with respect to the $\mathcal{DOS}$ semantics, and *not* with respect to the trimming semantics (which allows for arbitrary entry state in intermodule procedure calls). The. F.24 ensures soundness of the utilization of the procedure specification. Lem. F.31 and Lem. F.32 ensure that all possible input states are found.

Note that a *specification* of procedures of module m (Sec. 2) can be conservatively verified using any given fixpoint solution to the equation system shown in Fig. 11.

H Abstract Trimming Semantics

In this section, we define a *bounded* parametric abstraction for trimmed memory states. Our main goal in this section is to exemplify the effect of (one possible) bound abstraction on trimmed memory states.

We abstract sets of trimmed memory states by a point-wise application of an *extraction function* $\beta \colon \Sigma^\star \to \Sigma^{\star\sharp}$ (e.g., see [29]) mapping a trimmed memory state $\sigma^\star$ to its *best* representation by an *abstract trimmed memory state* $\sigma^{\star\sharp}$. We use set-union as the join-operator.

Intraprocedural

$$[\![n']\!]_\star = \bigcup_{\langle n,n'\rangle \in E_p} [\![n]\!]_\star \circ [\![stmt_{G_p}(\langle n,n'\rangle)]\!]_\star \qquad n' \neq s_p, n' \neq e_p, n' \text{ is not a return-site}$$

Interprocedural Intra-module (invocation of private procedures)

$$[\![s_p]\!]'_\star = \bigcup_{\langle n,n'\rangle \in E_q} \{\langle \sigma^{\star\prime\prime}, \sigma^{\star\prime\prime}\rangle \mid \langle \sigma^\star, \sigma^{\star\prime\prime}\rangle \in [\![n]\!]_\star \circ [\![Call_{y=p(x_1,\ldots,x_k)}]\!]_\star\} \qquad m(q) = m(p) = m,\ stmt_{G_p}(\langle n,n'\rangle) = invoke$$

$$[\![n']\!]_\star = [\![n]\!]_\star \circ \left\{ \langle \sigma^\star, \sigma^{\star\prime}\rangle \;\middle|\; \begin{array}{l} \sigma^{\star\prime} \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma^\star, \sigma^{\star\prime\prime}), \\ \sigma^{\star\prime\prime} \in [\![e_p]\!]_\star([\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma^\star)) \end{array} \right\} \qquad \begin{array}{l} \langle n,n'\rangle \in E_q, m(p) = m(q) = m \\ stmt_{G_p}(\langle n,n'\rangle) = invoke \end{array}$$

Interprocedural Inter-module (calls to lower modules)

$$[\![n']\!]_\star = [\![n]\!]_\star \circ \left\{ \langle \sigma^\star, \sigma^{\star\prime}\rangle \;\middle|\; \begin{array}{l} \sigma^{\star\prime} \in [\![Ret_{y=p(x_1,\ldots,x_k)}]\!]_\star(\sigma^\star, \sigma^{\star\prime\prime}), \\ \sigma^{\star\prime\prime} \in [\![p]\!]^s_\star([\![Call_{y=p(x_1,\ldots,x_k)}]\!](\sigma^\star)) \end{array} \right\} \qquad \begin{array}{l} \langle n,n'\rangle \in E_q, m(p) \neq m \\ stmt_{G_p}(\langle n,n'\rangle) = invoke \end{array}$$

Interprocedural Inter-module (simulating external calls to interface procedures)

$$[\![s_p]\!]_\star = \left\{ \langle \sigma^\star, \sigma^\star\rangle \;\middle|\; \begin{array}{c} c_i \in [\![t_p(z_i)]\!]_\star \text{ for } i = 1, \ldots, k, \\ \sigma^\star = \langle [z_i \mapsto hdr(c_i) \mid 1 \leq i \leq k], c_1 \oplus \ \ldots \oplus c_k \rangle \end{array} \right\} \qquad m(p) = m$$

Sealed microheaps

$$[\![T]\!]_\star = \bigcup_{m(q)=m} \left\{ c'|_l \;\middle|\; \begin{array}{c} \langle \sigma^\star, \sigma^{\star\prime}\rangle \in [\![e_p]\!]_\star,\ \sigma^{\star\prime} = \langle \rho', c'\rangle,\ t'(l) = T, \\ l \in \{\rho'(ret), \rho'(x) \mid x \in F_q\} \cap Loc \end{array} \right\} \qquad m(T) = m$$

Figure 11: Equation system whose (fixpoint) solution determines a sound module invariants for module m. We assume p is a procedure with k formal parameters, $z_1, \ldots, z_k$. $invoke \equiv y = p(x_1, \ldots, x_k)$. $c = \langle I, L, R, h, t, m\rangle$. We assume that $[\![T]\!]_\star = \{c^m_{rim}(l, T) \mid l \in Loc\}$ for all types T such that m uses $m(T)$. We denote the type of a local variable x of a procedure p by $t_p(x)$.

An abstract state $\sigma^{\star\sharp}$ provides a conservative bounded representation for the unbounded number of locations in every trimmed-state.

H.1 Parametric Abstract Domain

An abstract memory state $\sigma^\sharp = \langle \rho^\sharp, {}_M h^\sharp \rangle \in \Sigma^\sharp$ is comprised of an abstract environment $\rho^\sharp \in Env^\sharp$ of atomic (i.e., boolean) variables and an abstract current component $\mu h^\sharp \in \mu\mathcal{H}^\sharp$.

The abstraction of trimmed memory states is similar in flavor to the canonical abstraction of [40] 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 [33], we define an *abstract location* $l^\sharp$ to be a function from $Loc^\star_{prop}$ to $\{true, false\}$.

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 $l \in Loc$ is the pair of the location's properties and the type $type \in \mathcal{T}$ of the object at that location.

Given a *trimmed memory state*, we represent every concrete location l by an abstract location determined by the set of location properties l satisfies. This guarantees a bounded representation of the component. 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.

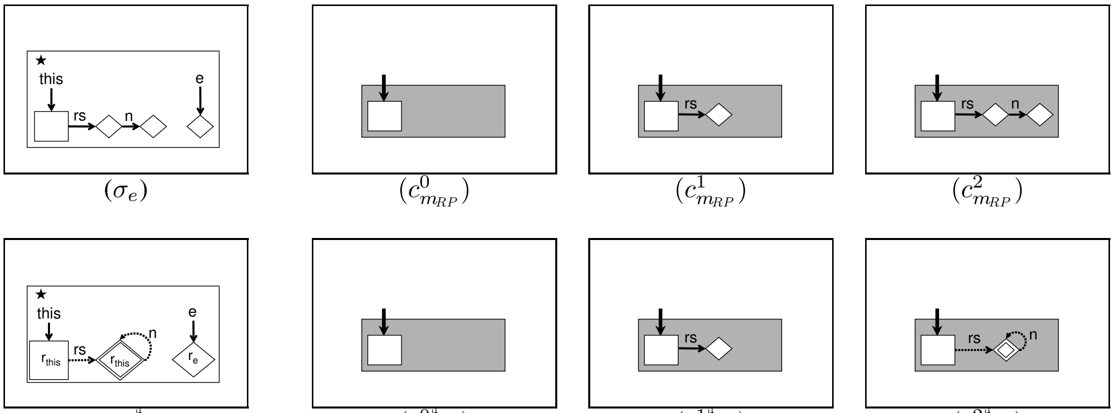
Mapping concrete locations to abstract locations induces a natural abstraction of the intra-microheap link-structure where an edge $\langle l^\sharp_s, f, l^\sharp_t \rangle$ means that the field f of a location l_s represented by $l^\sharp_s$ *may* point to a location l_t represented by $l^\sharp_t$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

Example H.1 Tab. 2 shows, as an example, the properties used in the analysis of the running example.[8] Fig. 12 depicts certain concrete and abstract trimmed memory states and sealed components. The abstraction is parameterized with the properties shown in Tab. 2. The graphical notation is like the one used for depicting concrete memory and sealed components in Fig. 3 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).
- A property of an abstract location is written inside the node depicting it. The absence of a property, indicates that the property does not hold for that abstract location. (The exception is that

Note that a list of two or more resources is represented by a single summary node. Also note that the absence of the labels ils and c from inside the depicted list nodes indicates that these lists are unshared and acyclic,

[8] In this paper, we do not address the issue of specifying the properties. One possible way to do so is to use first-order logic with transitive closure (FOTC). All the properties in Tab. 2 can be specified in FOTC.



(σ_e) $(c^0_{m_{RP}})$ $(c^1_{m_{RP}})$ $(c^2_{m_{RP}})$

$(\sigma_e{}^\sharp)$ $(c^{0\sharp}_{m_{RP}})$ $(c^{1\sharp}_{m_{RP}})$ $(c^{2\sharp}_{m_{RP}})$

Figure 12: Concrete and abstract trimmed memory states and sealed components. The abstraction of every memory state resp. component is drawn under it. (σ_e and $\sigma_e{}^\sharp$): The memory state that arise in our running example at the entry to `release` in the third invocation of `x.release(y)` and its abstraction, respectively. ($c^i_{m_{RP}}$ and $c^{i\sharp}_{m_{RP}}$ for $i = 0, 1, 2$): a sealed component of module m_{RP} with i resources in the pool and its abstraction, respectively. The abstract sealed component $c^{2\sharp}_{m_{RP}}$ conservatively represent any pool with 2 or more resources. Note that the abstract sealed component conservatively represent $[\![Inv_{m_{RP}}\ RPool]\!]$, the module invariant of type $RPool$ of module m_{RP}.

$Loc^\star_{prop}$	**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`?
$T(l)$	Is the object at location l of type T?
$ils(l)$	Is location l pointed-to by a field of more than 1 object inside the *microheap*?
$c(l)$	Does l reside on a directed cycle of fields?

Table 2: Abstraction parameters used in the running example. We use the properties in $Loc^\star_{prop}$ to represent the locations in all components of all modules.

respectively. Also note that the entry memory state is comprised of two disjoint parts: a resource pool component and a resource component. (The latter is not shown separately).

H.2 The Abstraction Function

Tab. 3 formally defines the β extraction function. Recall that the abstract domain, and the extraction function, are parametric in the recorded properties of locations inside components.

Parameters to the abstraction	Description
$l_{prop} \quad \in \quad Loc_{prop} = \{false, true\}^{\lvert Loc^{\star}_{prop} \rvert}$	Tracked properties of locations

Parameterized abstract domain	Description
$l^\sharp \quad \in \quad Loc^\sharp = Loc_{prop} \times \mathcal{T}$	Abstract locations
$v^\sharp \quad \in \quad Val^\sharp = Loc^\sharp \cup \{null\} \cup \{true, false\} \cup \{\ominus\}$	Abstract values
$t^\sharp \quad \in \quad \mathcal{T}^\sharp = Loc^\sharp \hookrightarrow \mathcal{T}$	Abstract types
$h^\sharp \quad \in \quad \mathcal{H}^\sharp = 2^{Loc^\sharp \times \mathcal{F} \times Val^\sharp}$	Abstract intra-component link structure
$c^\sharp \quad \in \quad \mathcal{C}^\sharp = 2^{Loc^\sharp} \times 2^{Loc^\sharp} \times 2^{Loc^\sharp} \times 2^{Loc^\sharp} \times \mathcal{H}^\sharp \times \mathcal{T}^\sharp \times \mathcal{M}$	Abstract components
$\rho^\sharp \quad \in \quad Env^\sharp = \mathcal{V} \hookrightarrow Val^\sharp$	Environment for non-pointer variables
$\sigma^{\star\sharp} \quad \in \quad \Sigma^{\star\sharp} = Env^\sharp \times \mathcal{C}^\sharp$	Abstract trimmed memory states

Figure 13: Parametric domain of abstract trimmed memory states. The domain is parameterized with $Loc^{\star}_{prop}$, the tracked properties of locations . $\{false, true\}^n$ is a boolean vector of length n.

$\beta \in \Sigma^\star \to \Sigma^{\star\sharp} \;\; s.t.$

$\beta(\langle \rho, \langle I, L, R, h, t, m \rangle \rangle) = \langle \rho^\sharp, \langle I^\sharp, L^\sharp, R^\sharp, SM^\sharp, h^\sharp, t^\sharp, m \rangle \rangle$ where:

$$\rho^\sharp(v) = \varphi \circ \rho$$
$$I^\sharp = \{\varphi(l) \mid l \in ip\}$$
$$L^\sharp = \{\varphi(l) \mid l \in L\}$$
$$R^\sharp = \{\varphi(l) \mid l \in R\}$$
$$SM^\sharp = \{l^\sharp \mid 1 < \lvert \{l \in L \cup R : \varphi(l) = l^\sharp\} \rvert \}$$
$$h^\sharp = \{\langle \varphi(l_s), f, \varphi(l_t) \rangle \mid h(l_s, f) = l_t \in Loc\} \cup \{\langle \varphi(l_s), f, v \rangle \mid h(l_s, f) = v \notin Loc\}$$
$$\varphi(v) = \begin{cases} \langle \phi(v), t(v) \rangle & v \in Loc \\ v & \text{otherwise} \end{cases}$$

Table 3: Parameterized extraction function.

The analysis encodes sets of properties of locations utilizing the *property vectors*[9] shown in Tab. 3. We assume to be given, for every module m, a property extraction function $l_{prop}(c, l) \in Loc_{prop}$ which maps every location l in every component c to their property vector.

The use of boolean vectors to encode the abstraction is inspired by [33].

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

I Extensions

Component-digests The trimming abstraction of parts of the heap outside the current component is very crude. Essentially, the types of the headers of the current component's subcomponents are recorded. We have extended our work to support richer abstractions. We allow to associate for every type t a bounded set of *component-digests*. Every digest can encode a possible typestate-like [42] property of a component and becomes part of the information recorded for that component in the rim of its owner (along with the type of the component's header). Every procedure that uses t should provide a specification of its effect on the digests of its parameters and the digest of the return value. This can allow our technique to distinguish between, say components pertaining to open or closed sockets, or a pool containing only closed sockets. Component digests are a matter of an ongoing work, thus, we delay further elaboration on this issue to a separate report.

Our work can be extended to allow for more sophisticated specifications (including possible nullness and aliasing of formal parameter, and digest-based specifications).

We note that our modular analysis is applicable even when the intra-module procedure calls are not cutpoint-free. However, we regard the issue of abstraction of local heaps with cutpoints as an orthogonal problem.

Inferring ownership specification As presented, our analysis requires user-supplied specification regarding ownership transfer. Actually, our analysis can infer a conservative specification on its own by detecting which of the objects which were passed as parameters always dominates its reachable heap at the exit site. Unfortunately, in many cases the result may not be useful. Consider the `release` procedure - when it returns, the resource parameter, and not the resource pool, dominates its reachable heap. There are different ways to change this unfortunate outcome. One is to use a heuristic to determine the transferable variables, e.g., the `this` variable (for object-oriented programs) or the return value always take precedence. Alternatively, our analysis can be trained to infer the specification by performing a whole-program analysis on example programs. Nevertheless, even the conservative specification it found can serve as a starting point to a manual specification effort, *e.g.*, the naive approach can detect the intended specification of `acquire`. In addition, given the definition of the digests, it is possible to automatically infer the speciation of the effect of inter-module procedure calls on the digests + refine the definition of modular invariant to be per digest.