Role Analysis

Viktor Kuncak, Patrick Lam, and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{vkuncak, plam, rinard}@lcs.mit.edu

ABSTRACT

We present a new *role system* in which the type (or *role*) of each object depends on its referencing relationships with other objects, with the role changing as these relationships change. Roles capture important object and data structure properties and provide useful information about how the actions of the program interact with these properties. Our role system enables the programmer to specify the legal aliasing relationships that define the set of roles that objects may play, the roles of procedure parameters and object fields, and the role changes that procedures perform while manipulating objects. We present an interprocedural, compositional, and context-sensitive role analysis algorithm that verifies that a program maintains role constraints.

1. INTRODUCTION

Types capture important properties of the objects that programs manipulate, increasing both the safety and readability of the program. Traditional type systems capture properties (such as the format of data items stored in the fields of the object) that are invariant over the lifetime of the object. But in many cases, properties that do change are as important as properties that do not. Recognizing the benefit of capturing these changes, researchers have developed systems in which the type of the object changes as the values stored in its fields change or as the program invokes operations on the object [45, 44, 10, 51, 52, 6, 18, 11]. These systems integrate the concept of changing object states into the type system.

The fundamental idea in this paper is that the type of each object should also depend on the data structures in which it participates. Our type system therefore captures the referencing relationships that determine this data structure participation. As objects move between data structures, their types change to reflect their changing relationships with other objects. Our system uses *roles* to formalize the concept of a type that depends on the referencing relationships. Each role declaration provides complete aliasing information for each object that plays that role—in addition to specifying roles for the fields of the object, the role declaration also identifies the complete set of references in the heap that refer to the object. In this way roles generalize linear type systems [48, 30] by allowing multiple aliases to be statically tracked, and extend alias types [43, 49] with the ability to specify the roles of objects that are the source of aliases.

This approach attacks a key difficulty associated with state-based type systems: the need to ensure that any state change performed using one alias is correctly reflected in the declared types of the other aliases. Because each object's role identifies all of its heap aliases, the analysis can verify the correctness of the role information at all remaining or new heap aliases after an operation changes the referencing relationships.

Roles capture important object and data structure properties, improving both the safety and transparency of the program. For example, roles allow the programmer to express data structure consistency properties (with the properties verified by the role analysis), to improve the precision of procedure interface specifications (by allowing the programmer to specify the role of each parameter), to express precise referencing and interaction behaviors between objects (by specifying verified roles for object fields and aliases), and to express constraints on the coordinated movements of objects between data structures (by using the aliasing information in the role definitions to identify legal data structure membership combinations). Roles may also aid program optimization by providing precise aliasing information.

This paper makes the following contributions:

- **Role Concept:** The concept that the state of an object depends on its referencing relationships; specifically, that objects with different heap aliases should be regarded as having different states.
- **Role Definition Language:** It presents a language for defining roles. The programmer can use this language to express data structure invariants and properties such as data structure participation.
- **Programming Model:** It presents a set of role consistency rules. These rules give a programming model for changing the role of an object and the circumstances under which roles can be temporarily violated.

*This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, and an NSERC graduate scholarship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL '02, Jan. 16-18, 2002 Portland, OR USA
Copyright 2002 ACM ISBN 1-58113-450-9/02/01 ...$5.00.

- **Procedure Interface Specification Language:** It presents a language for specifying the initial context and effects of each procedure. The effects summarize the actions of the procedure in terms of the references it changes and the regions of the heap that it affects.
- **Role Analysis Algorithm:** It presents an algorithm for verifying that the program respects the constraints given by a set of role definitions and procedure interface specifications. The algorithm uses a dataflow analysis similar to [41] to infer intermediate referencing relationships of objects, allowing the programmer to focus on role changes and procedure interfaces.

The remainder of the paper is organized as follows. Section 2 presents an example that introduces the concept of roles. Section 3 presents the semantics of roles as global heap invariants. Section 4 presents the programming model associated with roles. Section 5 presents the intraprocedural role analysis; Section 6 presents the interprocedural role analysis. Section 7 presents several extensions to the basic role framework. Section 8 presents the related work; we conclude in Section 9.

2. EXAMPLE

Figure 1 presents the *role reference diagram* for a process scheduler. Each box in the diagram denotes a disjoint set of objects that play a given role. The labelled arrows between boxes indicate possible references between the objects in each set. As the diagram indicates, the scheduler maintains a list of live processes. A live process can be either running or sleeping. The running processes form a doubly-linked list, while the sleeping processes form a binary tree. Both kinds of processes have a reference from the `proc` field of a live list node. The header objects `RunningHeader` and `SleepingHeader` simplify operations on the data structures that store the process objects.

As Figure 1 indicates, data structure participation determines the conceptual state of each object. In our example, processes that participate in the sleeping process tree are classified as sleeping processes, while processes that participate in the running process list are classified as running processes. Moreover, movements between data structures correspond to conceptual state changes—when a process stops sleeping and starts running, it moves from the sleeping process tree to the running process list.

2.1 Role Definitions

Figure 2 presents the role definitions in our example. Each role definition specifies the constraints that an object must satisfy to play the role. Field constraints specify the roles of the objects to which the fields refer, while slot constraints identify the number and kind of aliases of the object.[1]

In our example, the field constraints indicate that objects playing the `RunningProc` role have two fields: a `next` field that refers to either another `RunningProc` object or to the `RunningHeader` object, and a `prev` field that also refers to an object playing one of these two roles. The slot constraint indicates that `RunningProc` objects have three slots. The first slot is filled by a reference from the `next` field of either the `RunningHeader` object or a `RunningProc` object. The

[1]An alias of an object is a reference to that object stored in some field of some object in the heap.

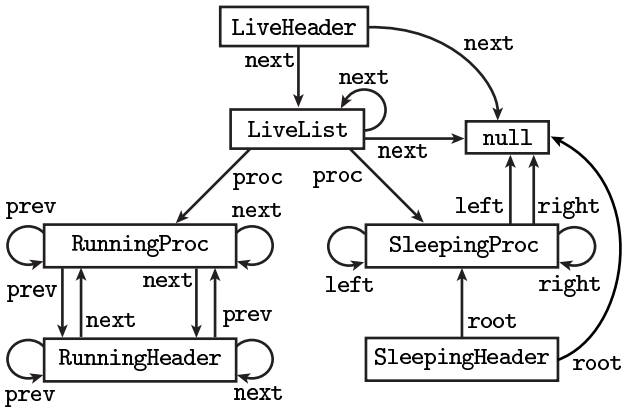

Figure 1: Role Reference Diagram for Scheduler

second slot is filled by a reference from the `prev` field of an object playing one of these two roles. The third and final slot is filled by a reference from the `proc` field of an object playing the `LiveList` role. In general, each slot must be filled by exactly one reference; together, these references make up the complete set of heap aliases of an object playing the role. In our example, this aliasing constraint implies that no process can be simultaneously running and sleeping.

Role definitions may also contain two additional kinds of constraints: identity constraints, which specify paths that lead back to the object, and acyclicity constraints, which specify paths with no cycles. In our example, the identity constraint `next.prev` in the `RunningProc` role specifies the cyclic doubly-linked list constraint that following the `next`, then `prev` fields always leads back to the initial object. The acyclic constraint `left, right` in the `SleepingProc` role specifies that there are no cycles in the heap involving only `left` and `right` edges. On the other hand, the list of running processes must be cyclic because its nodes can never point to `null`.

In general, roles can capture data structure consistency properties such as disjointness and can prevent representation exposure similarly to ownership types [8]. As a data structure description language, roles can naturally specify trees with additional pointers. Roles can also approximate non-tree data structures like sparse matrices. Because most role constraints are local, it is possible to inductively infer them from data structure instances.

2.2 Roles and Procedure Interfaces

Procedures specify the initial and final roles of their parameters. The `suspend` procedure in Figure 3, for example, takes two parameters: the `RunningProc` object `p`, and the `SleepingProc` object `s`. The procedure changes the role of the object referenced by `p` to `SleepingProc`, while the object referenced by `s` retains its original role. To perform the role change, the procedure removes `p` from the `RunningProc` linked list and inserts it into the `SleepingProc` tree `s`. If the procedure fails to perform the insertions or deletions correctly, for instance by leaving an object in both structures, the role analysis will report an error.

In addition to specifying the roles of the parameters, each procedure also specifies its read and write effects. For example, the `suspend` procedure specifies that it 1) must set the

```
role LiveHeader {
  fields next : LiveList | null;
}
role LiveList {
  fields next : LiveList | null,
         proc : RunningProc | SleepingProc;
  slots  LiveList.next | LiveHeader.next;
  acyclic next;
}
role RunningHeader {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev;
  identities next.prev, prev.next;
}
role RunningProc {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev,
         LiveList.proc;
  identities next.prev, prev.next;
}
role SleepingHeader {
  fields root : SleepingProc | null,
  acyclic left, right;
}
role SleepingProc {
  fields left  : SleepingProc | null,
         right : SleepingProc | null;
  slots SleepingProc.left | SleepingProc.right |
        SleepingHeader.root;
        LiveList.proc;
  acyclic left, right;
}
```

Figure 2: Role Definitions for Scheduler

`prev` and `next` fields of `p` to refer to `null`, 2) may set the `prev` and `next` fields of some objects playing the `RunningProc` or `RunningHeader` roles to refer to objects playing `RunningProc` or `RunningHeader` roles, and 3) must set the `root` field of `s` to refer to `p` and the `left` field of `p` to refer to either `null` or an object playing the `SleepingProc` role. Note that the `!` keyword, which precedes each must effect, syntactically distinguishes must effects (which the procedure must always perform) from may effects (which the procedure may or may not perform).

In general, each procedure may specify an initial context in the form of a graph that identifies procedure-specific referencing relationships. Nodes in this graph represent sets of objects in the heap; edges represent references between objects, and the context specifies a role for each node. The procedure specifies its effects at the granularity of the nodes in this graph. In our example, the procedure uses a default initial context which is derived automatically from the role reference diagram given the roles of the parameters.

2.3 Role Properties

In general, roles capture important properties of the objects and provide useful information about how the actions of the program affect those properties.

```
procedure suspend(p: RunningProc ->> SleepingProc,
                  s: SleepingHeader)
effects
  !(p.prev = null), !(p.next = null),
  (RunningProc|RunningHeader) . (prev|next) =
          (RunningProc|RunningHeader),
  !(s.root = p), !(p.left = SleepingProc | null);
var pp, pn, r;
{
   pp = p.prev;   pn = p.next;
   r = s.root;
   p.prev = null; p.next = null;
   pp.next = pn;  pn.prev = pp;
   s.root = p;    p.left = r;
   setRole(p : SleepingProc);
}
```

Figure 3: Suspend Procedure

- **Consistency Properties:** Roles can ensure that the program respects application-level data structure consistency properties. The roles in our process scheduler, for example, ensure that a process cannot be simultaneously sleeping and running.
- **Interface Changes:** In many cases, the interface of an object changes as its referencing relationships change. In our process scheduler, for example, only running processes can be suspended. Because procedures declare the roles of their parameters, the role system can ensure that the program uses objects correctly even as the object's interface changes.
- **Correlated Relationships:** In many cases, groups of objects cooperate to implement a piece of functionality. Standard type declarations provide some information about these collaborations by identifying the points-to relationships between related objects at the granularity of classes. But roles can capture a much more precise notion of cooperation, because they track correlated state changes of related objects.

3. SYNTAX AND SEMANTICS OF ROLES

In this section, we precisely define what it means for a given heap to satisfy a set of role definitions. In subsequent sections we will use this definition as a starting point for the programming model and role analysis.

3.1 Heap Representation

We represent a concrete program heap as a finite directed graph H_c with $\mathsf{nodes}(H_c)$ representing the objects in the heap and labelled edges representing heap references. A graph edge $\langle o_1, f, o_2 \rangle \in H_c$ denotes a reference with field name f from object o_1 to object o_2. To simplify the presentation, we fix a global set of fields F and assume that all objects have all fields in F.

3.2 Role Representation

Let R denote the set of roles used in role definitions, null_R be a special symbol always denoting a null object null_c, and $R_0 = R \cup \{\mathsf{null}_R\}$. We represent each role as the conjunction of the following four kinds of constraints:

- **Fields:** For every field name $f \in F$ we introduce a function $\mathsf{field}_f : R \to 2^{R_0}$ denoting the set of roles that objects of role $r \in R$ can reference through field f. A field f of role r can be null if and only if $\mathsf{null}_R \in \mathsf{field}_f(r)$. The explicit use of null_R and the possibility to specify a set of alternative roles for every field allows roles to express both may and must referencing relationships.

- **Slots:** Every role r has $\mathsf{slotno}(r)$ slots. A slot $\mathsf{slot}_k(r)$ of role $r \in R$ is a subset of $R \times F$. Let o be an object of role r and o' an object of role r'. A reference $\langle o', f, o\rangle \in H_c$ can fill slot number k of object o if and only if $\langle r', f\rangle \in \mathsf{slot}_k(r)$. An object with role r must therefore have exactly $\mathsf{slotno}(r)$ aliases.

- **Identities:** Every role $r \in R$ has a set $\mathsf{identities}(r) \subseteq F \times F$. Identities are pairs of fields $\langle f, g\rangle$ such that following reference f on object o and then returning on reference g leads back to o.

- **Acyclicities:** Every role $r \in R$ has a set $\mathsf{acyclic}(r) \subseteq F$ of fields along which cycles are forbidden.

The role definitions induce a role reference diagram RRD which captures some, but not all, of the role constraints.

$$\mathsf{RRD} = \{\langle r, f, r'\rangle \mid r' \in \mathsf{field}_f(r) \text{ and } \exists i\ \langle r, f\rangle \in \mathsf{slot}_i(r')\} \cup \{\langle r, f, \mathsf{null}_R\rangle \mid \mathsf{null}_R \in \mathsf{field}_f(r)\}$$

3.3 Role Semantics

We give the semantics of roles as a conjunction of invariants associated with the role definitions. A *concrete role assignment* is a map $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\rho_c(\mathsf{null}_c) = \mathsf{null}_R$.

DEFINITION 1. *Given a set of role definitions, we say that a heap H_c is* role consistent *iff there exists a role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that for every $o \in \mathsf{nodes}(H_c)$, the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ is satisfied. We call any such role assignment ρ_c a* valid *role assignment.*

The predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ formalizes the constraints associated with role definitions.

DEFINITION 2. $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ *iff all of the following conditions are met for $r = \rho_c(o)$.*

1) For every field $f \in F$, if $\langle o, f, o'\rangle \in H_c$ then $\rho_c(o') \in \mathsf{field}_f(r)$.

2) Let $\{\langle o_1, f_1\rangle, \ldots, \langle o_k, f_k\rangle\} = \{\langle o', f\rangle \mid \langle o', f, o\rangle \in H_c\}$ be the set of all aliases of object o. Then $k = \mathsf{slotno}(r)$ and there exists some permutation p of $\{1, \ldots, k\}$ such that $\langle \rho_c(o_i), f_i\rangle \in \mathsf{slot}_{p_i}(r)$ for all i.

3) If $\langle o, f, o'\rangle \in H_c$, $\langle o', g, o''\rangle \in H_c$, and $\langle f, g\rangle \in \mathsf{identities}(r)$, then $o = o''$.

4) It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ where $o_1 = o$ and $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$

Note that a role consistent heap may have multiple valid role assignments ρ_c. However, in each of these role assignments, every object o is assigned exactly one role $\rho_c(o)$. The existence of a role assignment ρ_c with the property $\rho_c(o_1) \neq \rho_c(o_2)$ thus implies $o_1 \neq o_2$.

4. A PROGRAMMING MODEL

In this section we define what it means for an execution of a program to respect the role constraints. Our goal is to allow the program to temporarily violate the constraints during data structure manipulations. To achieve this goal, we let the program violate the constraints for objects referenced by local variables or parameters, but require all other objects to satisfy the constraints.

We assume a simple imperative language with dynamic object allocation. The language contains, as basic statements, Load (`x=y.f`), Store (`x.f=y`), Copy (`x=y`), and New (`x=new`). We use a `test` statement combined with nondeterministic choice and iteration to express `if` and `while` statement, using the standard translation [20, 3]. We represent the control flow of programs using control-flow graphs.

A program is a collection of procedures $\mathsf{proc} \in \mathsf{Proc}$. Procedures change the global heap but do not return values. Every procedure proc has a list of parameters $\mathsf{param}(\mathsf{proc}) = \{\mathsf{param}_i(\mathsf{proc})\}_i$ and a list of local variables $\mathsf{locals}(\mathsf{proc})$. A procedure definition specifies the initial role $\mathsf{preR}_k(\mathsf{proc})$ and the final role $\mathsf{postR}_k(\mathsf{proc})$ for every parameter $\mathsf{param}_k(\mathsf{proc})$. We use proc_j for indices $j \in \mathcal{N}$ to denote the activation records of procedure proc. We represent a local variable or a parameter x as an edge $\langle \mathsf{proc}_j, x, o\rangle \in H_c$ from the activation record proc_j to the referenced object o. We further assume that there are no modifications of parameter variables so that every parameter references the same object throughout the procedure execution. We use an operational semantics with explicit error states that have heap error_c.

4.1 Onstage and Offstage Objects

At every program point the set of all heap objects can be partitioned into:

1. **onstage objects** $\mathsf{onstage}(H_c)$ referenced by a local variable or parameter of some activation record;
2. **offstage objects** $\mathsf{offstage}(H_c)$ unreferenced by local or parameter variables.

Onstage objects need not have correct roles. Offstage objects must have almost correct roles assuming some role assignment for onstage objects.

DEFINITION 3. *Given a set of role definitions and a set of objects $S_c \subseteq \mathsf{nodes}(H_c)$, we say that a heap H_c is* role consistent for S_c, *and we write $\mathsf{con}(H_c, S_c)$, iff there exists a role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ is satisfied for all $o \in S_c$.*

We define $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ to generalize the $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ predicate, weakening the acyclicity condition.

DEFINITION 4. $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ *holds iff conditions 1), 2), and 3) of Definition 2 are satisfied and the following condition holds:*

4') It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ such that $o_1 = o$, $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$, and additionally $o_1, \ldots, o_s \in S_c$.

Here S_c is the set of onstage objects that are not allowed to create a cycle; the objects in $\mathsf{nodes}(H_c) \setminus S_c$ are exempt from the acyclicity condition. The $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$

and $\mathsf{con}(H_c, S_c)$ predicates are monotonic in S_c, so a larger S_c implies a stronger invariant. For $S_c = \mathsf{nodes}(H_c)$, consistency for S_c is equivalent to heap consistency from Definition 1. Note that the role assignment ρ_c specifies roles even for objects $o \in \mathsf{nodes}(H_c) \setminus S_c$. This is because the role of o may influence the role consistency of objects in S_c which are adjacent to o.

At procedure calls, the role declarations for parameters restrict the set of potential role assignments. We therefore generalize $\mathsf{con}(H_c, S_c)$ to $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, which restricts the set of role assignments ρ_c considered for heap consistency.

DEFINITION 5. *Given a set of role definitions, a heap H_c, a set $S_c \subseteq \mathsf{nodes}(H_c)$, and a partial role assignment $\mathsf{ra} \subseteq S_c \to R$, we say that the heap H_c is* consistent with ra for S_c, *and write $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, iff there exists a (total) role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\mathsf{ra} \subseteq \rho_c$ and for every object $o \in S_c$ the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ is satisfied.*

4.2 Role Consistency

We are now able to precisely state the role consistency requirements that must be satisfied for program execution. We extend the operational semantics of the language with transitions leading to a program state with heap error_c whenever one of the constraints below is violated.

4.2.1 Offstage Consistency

At every program point, we require $\mathsf{con}(H_c, \mathsf{offstage}(H_c))$ to be satisfied. This means that offstage objects have correct roles, but onstage objects may have their role temporarily violated.

4.2.2 Reference Removal Consistency

The Store statement `x.f=y` has the following safety precondition. When removing a reference $\langle o_x, f, o_f \rangle \in H_c$ for $\langle \mathsf{proc}_j, \mathtt{x}, o_x \rangle \in H_c$, and $\langle o_x, \mathtt{f}, o_f \rangle \in H_c$ from the heap, both o_x and o_f must be onstage. It is sufficient to verify this condition for o_f, because o_x is already onstage by definition. The reference removal consistency condition enables the completion of the role change for o_f after the reference $\langle o_x, f, o_f \rangle$ is removed and ensures that heap references are introduced and removed only between onstage objects.

4.2.3 Procedure Call Consistency

Our programming model ensures role consistency across procedure calls using the following protocol.

A procedure call $\mathsf{proc}'(x_1, ..., x_p)$ requires the role consistency precondition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, where the partial role assignment ra requires objects o_i, corresponding to parameters x_i, to have roles $\mathsf{preR}_i(\mathsf{proc}')$ expected by the callee, where $S_c = \mathsf{offstage}(H_c) \cup \{o_i\}_i$ for $\langle \mathsf{proc}_j, x_i, o_i \rangle \in H_c$.

To ensure that the callee proc'_j never observes incorrect roles, we impose an *accessibility condition* for the callee's Load statements. The accessibility condition prohibits access to any object o referenced by some local variable of an activation record other than proc'_j, unless o is referenced by some parameter of proc'_j. Provided that this condition is not violated, the callee proc'_j only accesses objects with correct roles, even though objects that it does not access may have incorrect roles. The role analysis uses read effects (Section *6.1.2*) to ensure that the accessibility condition is never violated.

At the procedure exit point, we require correct roles for all objects referenced by the current activation record proc'_j. This implies that heap operations performed by proc'_j preserve heap consistency for all objects accessed by proc'_j.

4.3 Instrumented Semantics

We expect the programmer to have a specific role assignment in mind when writing the program; this role assignment changes as the statements of the program change the referencing relationships. Thus, when the programmer wishes to change the role of an object, he or she writes a program that brings the object onstage, changes its referencing relationships so that it plays a new role, then puts it offstage in its new role. The roles of other objects do not change.[2]

To support these programmer expectations, we introduce an augmented programming model in which the role assignment is conceptually part of the program's state and changes under the control of the programmer. This augmented model has an underlying *instrumented semantics* as opposed to the *original semantics*.

The instrumented semantics extends the concrete heap H_c with a role assignment ρ_c. We introduce a new statement `setRole(x:r)`, which modifies a role assignment ρ_c, giving $\rho_c[o_x \mapsto r]$, where o_x is the object referenced by `x`. All statements other than `setRole` preserve the current role assignment. For every consistency condition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$ in the original semantics, the instrumented semantics uses the corresponding condition $\mathsf{conW}(\rho_c \cup \mathsf{ra}, H_c, S_c)$ and fails if ρ_c is not an extension of ra.

Note that our instrumented semantics does not imply an implementation that explicitly stores roles of objects. We instead use the instrumented semantics as the basis of our role analysis and ensure that all role checks can be statically removed. Because the instrumented semantics is more restrictive than the original semantics, our role analysis is a conservative approximation of both the instrumented semantics and the original semantics.

5. INTRAPROCEDURAL ROLE ANALYSIS

Our analysis representation is a graph in which nodes represent objects and edges represent references between objects. There are two kinds of nodes: *onstage nodes* represent onstage objects, with each onstage node representing one onstage object; and *offstage nodes*, with each offstage node representing a set of objects that play that role. To increase the precision of the analysis, the algorithm occasionally generates multiple offstage nodes that represent disjoint sets of objects playing the same role. The goal is to capture reachability distinctions; offstage objects that have the same role but are represented by different offstage nodes have different reachability properties from onstage nodes.

The key observation behind our analysis algorithm is that role consistency for the concrete heap H_c can be verified incrementally by ensuring role consistency for every node when it goes offstage. This allows us to represent the statically unbounded offstage portion of the heap using summary nodes with "may" references. In contrast, we use a "must" interpretation for references to and from onstage

[2] An extension to the programming model supports *cascading role changes* in which a single role change propagates through the heap changing the roles of offstage objects, see Section 7.1.

nodes, which allows the analysis to verify role consistency in the presence of temporary violations of role constraints.

We frame the role analysis as a dataflow analysis operating on a distributive lattice $\mathcal{P}(\mathsf{RoleGraphs})$ of sets of role graphs with set union $\cup$ as the join operator. In this section we present an algorithm for intraprocedural analysis. We use proc_c to denote the topmost activation record in a concrete heap H_c. More details on the intraprocedural role analysis can be found in [32].

5.1 Abstraction Relation

Every dataflow fact $\mathcal{G} \subseteq \mathsf{RoleGraphs}$ is a set of role graphs $G \in \mathcal{G}$. Every role graph $G \in \mathsf{RoleGraphs}$ is either a bottom role graph $\bot_G$ representing the set of all concrete heaps (including error_c), or a tuple $G = \langle H, \rho, K \rangle$ representing non-error concrete heaps, where

- $H \subseteq N \times F \times N$ is the abstract heap with nodes N (representing objects) and fields F. The abstract heap H represents heap references $\langle n_1, f, n_2 \rangle$ and variables of the currently analyzed procedure $\langle \mathsf{proc}, x, n \rangle$ where $x \in \mathsf{locals}(\mathsf{proc})$. Null references are represented as references to abstract node null. We define two sets of abstract nodes: onstage nodes $\mathsf{onstage}(H) = \{n \mid \langle \mathsf{proc}, x, n \rangle \in H, x \in \mathsf{locals}(\mathsf{proc}) \cup \mathsf{param}(\mathsf{proc}), n \neq \mathsf{null}\}$ and offstage nodes $\mathsf{offstage}(H) = \mathsf{nodes}(H) \setminus \mathsf{onstage}(H) \setminus \{\mathsf{proc}, \mathsf{null}\}$.
- $\rho : \mathsf{nodes}(H) \to R_0$ is an abstract role assignment, $\rho(\mathsf{null}) = \mathsf{null}_R$;
- $K : \mathsf{nodes}(H) \to \{i, s\}$ indicates the kind of each node; when $K(n) = i$, then n is an individual node representing at most one object, and when $K(n) = s$, n is a summary node representing zero or more objects. We require $K(\mathsf{proc}) = K(\mathsf{null}) = i$, and require all onstage nodes to be individual, $K[\mathsf{onstage}(H)] \subseteq \{i\}$.

The abstraction relation α relates a pair $\langle H_c, \rho_c \rangle$ of a concrete heap and a concrete role assignment to an abstract role graph G.

DEFINITION 6. *We say that an abstract role graph G represents a concrete heap H_c with role assignment ρ_c and write $\langle H_c, \rho_c \rangle \, \alpha \, G$, iff $G = \bot_G$ or: $H_c \neq \mathsf{error}_c$, $G = \langle H, \rho, K \rangle$, and there exists a function $h : \mathsf{nodes}(H_c) \to \mathsf{nodes}(H)$ such that*

1) *H_c is role consistent: $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$,*
2) *identity constraints of onstage nodes with offstage nodes hold: if $\langle o_1, f, o_2 \rangle \in H_c$ and $\langle o_2, g, o_3 \rangle \in H_c$ for $o_1 \in \mathsf{onstage}(H_c)$, $o_2 \in \mathsf{offstage}(H_c)$, and $\langle f, g \rangle \in \mathsf{identities}(\rho_c(o_1))$, then $o_3 = o_1$;*
3) *h is a graph homomorphism: if $\langle o_1, f, o_2 \rangle \in H_c$ then $\langle h(o_1), f, h(o_2) \rangle \in H$;*
4) *an individual node represents at most one concrete object: $K(n) = i$ implies $|h^{-1}(n)| \leq 1$;*
5) *h is bijection on edges which originate or terminate at onstage nodes: if $\langle n_1, f, n_2 \rangle \in H$ and $n_1 \in \mathsf{onstage}(H)$ or $n_2 \in \mathsf{onstage}(H)$, then there exists exactly one $\langle o_1, f, o_2 \rangle \in H_c$ such that $h(o_1) = n_1$ and $h(o_2) = n_2$;*
6) *$h(\mathsf{null}_c) = \mathsf{null}$ and $h(\mathsf{proc}_c) = \mathsf{proc}$;*
7) *the abstract role assignment ρ corresponds to the concrete role assignment: $\rho_c(o) = \rho(h(o))$ for every object $o \in \mathsf{nodes}(H_c)$.*

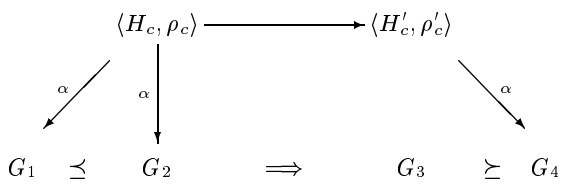

Figure 4: Simulation Relation Between Abstract and Concrete Semantics

Note that the error heap error_c can be represented only by the bottom role graph $\bot_G$. The analysis uses $\bot_G$ to indicate a potential role error.

Condition 3) implies that role graph edges are a conservative approximation of concrete heap references. These edges are in general "may" edges. Hence it is possible for an offstage node n that $\langle n, f, n_1 \rangle, \langle n, f, n_2 \rangle \in H$ for $n_1 \neq n_2$. This cannot happen when $n \in \mathsf{onstage}(H)$ because of 5). Another consequence of 5) is that the existence of an edge in H from an onstage node n_0 to a summary node n_s implies that n_s represents at least one object. Condition 2) strengthens 1) by requiring certain identity constraints for onstage nodes to hold, as explained in Section *5.2.4*.

5.2 Transfer Functions

The key complication in developing the transfer functions for the role analysis is to accurately model the movement of objects onstage and offstage. For example, a load statement `x=y.f` may cause the object referred to by `y.f` to move onstage. In addition, if `x` was the only reference to an onstage object o before the statement executed, object o moves offstage after the execution of the load statement, and thus must satisfy the locallyConsistent predicate.

The analysis uses an expansion relation $\preceq$ to model the movement of objects onstage and a contraction relation $\succeq$ to model the movement of objects offstage. Conceptually, the expansion relation pulls onstage nodes out of offstage nodes as necessary to model the actions of each statement. The contraction relation merges onstage nodes into offstage nodes to model the movement of objects offstage.

We present our role analysis as an abstract execution relation $\rightsquigarrow$. The abstract execution uses expansion and contraction to ensure that the abstraction relation α is a forward simulation relation from the space of concrete heaps with role assignments to the set RoleGraphs. The simulation relation implies that the traces of $\rightsquigarrow$ include the traces of the instrumented semantics $\longrightarrow$. To make sure that the program does not violate the role constraints, it is thus sufficient to guarantee that $\bot_G$ is not reachable via $\rightsquigarrow$.

To prove that $\bot_G$ is not reachable in the abstract execution, the analysis computes, for every program point p, a set of role graphs $\mathcal{G}$ that conservatively approximates the possible program states at point p. The transfer function for a statement st is given as an image $[\![\mathsf{st}]\!](\mathcal{G}) = \{G' \mid G \in \mathcal{G}, G \overset{\mathtt{st}}{\rightsquigarrow} G'\}$. The analysis computes the relation $\overset{\mathtt{st}}{\rightsquigarrow}$ in three steps: 1) ensure that the relevant nodes are instantiated using the expansion relation $\preceq$; 2) perform the symbolic execution $\overset{\mathtt{st}}{\Longrightarrow}$ of the statement `st`; 3) if needed, merge nodes using the contraction relation $\succeq$ to keep the role graph bounded. Figure 4 shows how the abstraction relation α relates $\preceq$,

$\stackrel{\mathtt{st}}{\Longrightarrow}$, and $\succeq$ with the concrete execution $\longrightarrow$ in the instrumented semantics. One of the role graphs G_2 obtained after expansion remains an abstraction of $\langle H_c, \rho_c \rangle$. The symbolic execution of a statement followed by the contraction relation corresponds to a step in the instrumented operational semantics.

Figure 5 presents the definition of the abstract execution relation $\stackrel{\mathtt{st}}{\rightsquigarrow}$. Only the Load statement uses the expansion relation, because the other statements operate on objects that are already onstage. The Load, Copy, and New statements may remove a local variable reference from an object, so they use the contraction relation to move the object offstage if needed. For the rest of the statements, the abstract execution reduces to the symbolic execution $\Longrightarrow$ described in Section *5.2.3*.

5.2.1 Expansion

Given a role graph $\langle H, \rho, K \rangle$, the expansion relation $\stackrel{n,f}{\preceq}$ (presented in Figure 6) attempts to produce a set of role graphs $\langle H', \rho', K' \rangle$ in each of which $\langle n, f, n_0 \rangle \in H'$ and $K(n_0) = i$. The analysis uses the expansion in the abstract execution of the Load statement. The expansion first performs a check for null pointer dereference and reports an error if the check fails. If $\langle n, f, n' \rangle \in H$ and n' is onstage, the expansion returns the original state. Otherwise, the expansion first instantiates the offstage node n' using the instantiation relation $\Uparrow$.

The instantiation relation (Figure 7) generates a set of role graphs which approximate the original role graph and have n_0 as an onstage node. The instantiation uses localCheck (Section *5.2.4*) to filter out cases which cannot occur given the role constraints.

The analysis next applies the split operation $\|$ (see [32]) if the acycCheck (Section *5.2.4*) does not hold for the newly instantiated node. Let $\rho(n_0) = r$. The split operation ensures that in the role graph n_0 is not a member of any cycle of offstage nodes which contains only edges in $\mathsf{acyclic}(r)$ by splitting the offstage nodes. In this way split encodes the reachability information implied by the acyclicity conditions. The analysis can use this information even after the role of node n_0 changes. In particular, this allows the verification of the acyclicity condition for n_0 when n_0 moves offstage.

5.2.2 Contraction

The analysis applies the contraction relation $\stackrel{n}{\succeq}$ (Figure 8) when a local variable reference to node n is removed. If there are other local variables referencing n, contraction does nothing. Otherwise n has just gone offstage, so contraction invokes nodeCheck (Section *5.2.4*) to ensure that the role of n is consistent with its referencing relationships. If the check fails, the result is $\perp_G$. If the check succeeds, the contraction applies the normalization operation to ensure that the role graph remains bounded.

The normalization operation takes a role graph $\langle H, \rho, K \rangle$ and produces a role graph $\langle H', \rho', K' \rangle$ that is a factor graph of the original graph under the equivalence relation $\sim$. Two offstage nodes are equivalent under $\sim$ if they have the same role and the same may reachability from all onstage nodes. Here we consider n to be reachable from an onstage node n_0 if there is some path in the role graph from n_0 to n whose edges belong to $\mathsf{acyclic}(\rho(n_0))$ and whose nodes are all in $\mathsf{offstage}(H)$. In this way normalization avoids merging nodes which were previously generated in the split operation $\|$, while still ensuring a bound on the size of the role graph.

5.2.3 Symbolic Execution

The symbolic execution $\stackrel{\mathtt{st}}{\Longrightarrow}$ of most statements st in Figure 9 acts on the abstract heap in the same way that the statement would act on the concrete heap. In particular, the Store statement always performs strong updates. The simplicity of the symbolic execution is due to conditions 3) and 5) in the abstraction relation α. (The analysis maintains these conditions using the expansion relation $\preceq$.) The symbolic execution also verifies the consistency conditions that are not verified by $\preceq$ or $\succeq$.

5.2.3.1 Verifying Reference Removal Consistency.

The transfer relation $\stackrel{\mathtt{st}}{\rightsquigarrow}$ for the Store statement can easily verify the Store safety condition from section *4.2.2*, because the set of onstage and offstage nodes is known precisely for every role graph. It returns $\perp_G$ if the safety condition does not hold.

5.2.3.2 Symbolic Execution of setRole.

The `setRole(x:r)` statement sets the role of node n_x referenced by the variable `x` to `r`. Let $G = \langle H, \rho, K \rangle$ be the current role graph and let $\langle \mathsf{proc}, \mathtt{x}, n_x \rangle \in H$. If n_x has no adjacent offstage nodes, the role change always succeeds. In general, there are restrictions on when the change can be done. Let $\langle H_c, \rho_c \rangle$ be a concrete heap with role assignment represented by G and h be a homomorphism from H_c to H. Let $h(o_x) = n_x$ and $r_0 = \rho_c(o_x)$. The symbolic execution must make sure that the condition $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$ continues to hold after the role change. Because the set of onstage objects does not change, it suffices to ensure that the original roles for offstage nodes are consistent with the new role r. The acyclicity constraint involves only offstage objects, so it remains satisfied. The other role constraints are local, so they can only be violated for offstage neighbors of o_x. To make sure that no violations occur, we require:

1. $\mathtt{r} \in \mathsf{field}_f(\rho(n))$ for all $\langle n, f, n_x \rangle \in H$, and
2. $\langle \mathtt{r}, f \rangle \in \mathsf{slot}_i(\rho(n))$ for all $\langle n_x, f, n \rangle \in H$ and every slot i such that $\langle r_0, f \rangle \in \mathsf{slot}_i(\rho(n))$

This is sufficient to guarantee $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$. To ensure condition 2) in Definition 6 of the abstraction relation, we require that for every $\langle f, g \rangle \in \mathsf{identities}(\mathtt{r})$,

1. $\langle f, g \rangle \in \mathsf{identities}(r_0)$ or
2. for all $\langle n_x, f, n \rangle \in H$: $K(n) = i$ and ($\langle n, g, n' \rangle \in H$ implies $n' = n_x$).

We use $\mathsf{roleChOk}(n_x, \mathtt{r}, \langle H, \rho, K \rangle)$ to denote the check just described.

5.2.4 Checking Node Properties

The analysis uses the localCheck, acycCheck, acycCheckAll, and nodeCheck predicates to incrementally maintain the abstraction relation.

We first define the predicate localCheck, which roughly corresponds to the predicate locallyConsistent (Definition 2), but ignores the nonlocal acyclicity condition and additionally ensures condition 2) from Definition 6. Our role analysis

Transition	Definition	Conditions
$\langle H,\rho,K\rangle \overset{\texttt{x=y.f}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{n_y,f}{\preceq} G_1 \overset{\texttt{x=y.f}}{\Longrightarrow} G_2 \overset{n_x}{\succeq} G'$	$\langle \mathsf{proc},\mathtt{x},n_x\rangle, \langle \mathsf{proc},\mathtt{y},n_y\rangle \in H$
$\langle H,\rho,K\rangle \overset{\texttt{x=y}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{\texttt{x=y}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc},\mathtt{x},n_1\rangle \in H$
$\langle H,\rho,K\rangle \overset{\texttt{x=new}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{\texttt{x=new}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc},\mathtt{x},n_1\rangle \in H$
$\langle H,\rho,K\rangle \overset{\mathtt{s}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} G'$	$s \in \{\texttt{x.f=y}, \texttt{test(c)}, \texttt{setRole(x:r)}\}$

Figure 5: Abstract Execution Relation $\rightsquigarrow$

Transition	Definition	Condition
$\langle H,\rho,K\rangle \overset{n,f}{\preceq} \langle H,\rho,K\rangle$		$\langle n,f,n'\rangle \in H, n' \in \mathsf{onstage}(H)$
$\langle H,\rho,K\rangle \overset{n,f}{\preceq} G'$	$\langle H,\rho,K\rangle \overset{n_0}{\underset{n'}{\Uparrow}} \langle H_1,\rho_1,K_1\rangle \overset{n_0}{\|} G'$	$\langle n,f,n'\rangle \in H, n' \in \mathsf{offstage}(H)$ $\langle n,f,n_0\rangle \in H_1$

Figure 6: Expansion Relation $\preceq$

$\langle H,\rho,K\rangle \overset{n_0}{\underset{n'}{\Uparrow}} \langle H',\rho',K'\rangle$	$H' = H \setminus H_0 \cup H_0' \cup H_1'$ $n' \notin \mathsf{nodes}(H'),$ if $K(n') = i$ $\rho' = \rho[n_0 \mapsto \rho(n')]$ $K' = K[n_0 \mapsto i]$ $\mathsf{localCheck}(n_0, \langle H',\rho',K'\rangle)$ $H_0 \subseteq H \cap \big(\mathsf{onstage}(H) \times F \times \{n'\} \cup \{n'\} \times F \times \mathsf{onstage}(H)\big)$ $H_1 \subseteq H \cap \big(\mathsf{offstage}(H) \times F \times \{n'\} \cup \{n'\} \times F \times \mathsf{offstage}(H)\big)$ $H_0' = \mathsf{swing}(n', n_0, H_0)$ $H_1' \subseteq \mathsf{swing}(n', n_0, H_1)$

$$\begin{aligned}\mathsf{swing}(n_{\mathsf{old}}, n_{\mathsf{new}}, H) \quad = \quad & \{\langle n_{\mathsf{new}}, f, n\rangle \mid \langle n_{\mathsf{old}}, f, n\rangle \in H\} \cup \{\langle n, f, n_{\mathsf{new}}\rangle \mid \langle n, f, n_{\mathsf{old}}\rangle \in H\} \cup \\ & \{\langle n_{\mathsf{new}}, f, n_{\mathsf{new}}\rangle \mid \langle n_{\mathsf{old}}, f, n_{\mathsf{old}}\rangle \in H\}\end{aligned}$$

Figure 7: Instantiation Relation $\Uparrow$

$\langle H,\rho,K\rangle \overset{n}{\succeq} \langle H,\rho,K\rangle$	$\exists \mathtt{x}\ \langle \mathsf{proc},\mathtt{x},n\rangle \in H$
$\langle H,\rho,K\rangle \overset{n}{\succeq} \mathsf{normalize}(\langle H,\rho,K\rangle)$	$\mathsf{nodeCheck}(n, \langle H,\rho,K\rangle, \mathsf{offstage}(H))$

Figure 8: Contraction Relation $\succeq$

Statement s	Transition	Conditions
`x = y.f`	$\langle H \uplus \{\mathsf{proc},\mathtt{x},n_x\},\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc},\mathtt{x},n_f\},\rho,K\rangle$	$\langle \mathsf{proc},\mathtt{y},n_y\rangle, \langle n_y,f,n_f\rangle \in H$
`x.f = y`	$\langle H \uplus \{n_x,f,n_f\},\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} \langle H \uplus \{n_x,f,n_y\},\rho,K\rangle$	$\langle \mathsf{proc},\mathtt{x},n_x\rangle, \langle \mathsf{proc},\mathtt{y},n_y\rangle \in H$ $n_f \in \mathsf{onstage}(H)$
`x = y`	$\langle H \uplus \{\mathsf{proc},\mathtt{x},n_x\},\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc},\mathtt{x},n_y\},\rho,K\rangle$	$\langle \mathsf{proc},\mathtt{y},n_y\rangle \in H$
`x = new`	$\langle H \uplus \{\mathsf{proc},\mathtt{x},n_x\},\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc},\mathtt{x},n_n\},\rho',K\rangle$	n_n fresh $\rho' = \rho[n_n \mapsto \mathsf{unknown}]$
`test(c)`	$\langle H,\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} \langle H,\rho,K\rangle$	$\mathsf{satisfied}(\mathtt{c}, H)$
`setRole(x:r)`	$\langle H,\rho,K\rangle \overset{\mathtt{s}}{\Longrightarrow} \langle H,\rho[n_x \mapsto \mathtt{r}],K\rangle$	$\langle \mathsf{proc},\mathtt{x},n_x\rangle \in H$ $\mathsf{roleChOk}(n_x,\mathtt{r},\langle H,\rho,K\rangle)$

$\mathsf{satisfied}(\texttt{x==y}, H_c)$ iff $\{o \mid \langle \mathsf{proc},\mathtt{x},o\rangle \in H_c\} = \{o \mid \langle \mathsf{proc},\mathtt{y},o\rangle \in H_c\}$

$\mathsf{satisfied}(\texttt{!(x==y)}, H_c)$ iff not $\mathsf{satisfied}(\texttt{x==y}, H_c)$

Figure 9: Symbolic Execution of Basic Statements $\Longrightarrow$

uses localCheck in the instantiation relation (Section *5.2.1*) and in the contraction relation (Section *5.2.2*).

DEFINITION 7. *For a role graph $G = \langle H, \rho, K\rangle$, an individual node n and a set S, the predicate* $\mathsf{localCheck}(n, G)$ *holds iff the following conditions are met for $r = \rho(n)$.*

1A. (Outgoing fields check) For fields $f \in F$, if $\langle n, f, n'\rangle \in H$ then $\rho(n') \in \mathsf{field}_f(r)$.

2A. (Incoming slots check) Let $\{\langle n_1, f_1\rangle, \ldots, \langle n_k, f_k\rangle\} = \{\langle n', f\rangle \mid \langle n', f, n\rangle \in H\}$ be the set of all aliases of node n in abstract heap H. Then $k = \mathsf{slotno}(r)$ and there exists a permutation p of the set $\{1, \ldots, k\}$ such that $\langle \rho(n_i), f_i\rangle \in \mathsf{slot}_{p_i}(r)$ for all i.

3A. (Identity Check) If $\langle n, f, n'\rangle \in H$, $\langle n', g, n''\rangle \in H$, $\langle f, g\rangle \in \mathsf{identities}(r)$, and $K(n') = i$, then $n = n''$.

4A. (Neighbor Identity Check) For every edge $\langle n', f, n\rangle \in H$, if $K(n') = i$, $\rho(n') = r'$ and $\langle f, g\rangle \in \mathsf{identities}(r')$ then $\langle n, g, n'\rangle \in H$.

5A. (Field Sanity Check) For every $f \in F$ there is exactly one edge $\langle n, f, n'\rangle \in H$.

Conditions 1A and 2A correspond to conditions 1) and 2) in Definition 2. Condition 3) in Definition 4 is not necessarily implied by condition 3A) if some of the neighbors of n are summary nodes. Condition 3) cannot be established based only on summary nodes, because verifying an identity constraint for field f of node n where $\langle n, f, n'\rangle \in H$ requires knowing the identity of n', not only its existence and role. We therefore rely on Condition 2) of Definition 6 to ensure that the identity relations of neighbors of node n are satisfied before n moves offstage.

The predicate $\mathsf{acycCheck}(n, G, S)$ verifies the acyclicity condition from Definition 4 for the node that has just been brought onstage. The split operation uses acycCheck to check whether it should split any nodes (Section *5.2.1*). The set S represents offstage nodes.

DEFINITION 8. *We say that a node n satisfies an acyclicity check in graph $G = \langle H, \rho, K\rangle$ with respect to the set S, and we write* $\mathsf{acycCheck}(n, G, S)$, *iff it is not the case that H contains a cycle $n_1, f_1, \ldots, n_s, f_s, n_1$ where $n_1 = n$, $f_1, \ldots, f_s \in \mathsf{acyclic}(\rho(n))$ and $n_1, \ldots, n_s \in S$.*

The analysis uses the predicate $\mathsf{acycCheckAll}(n, G, S)$ in the contraction relation (Section *5.2.2*) to make sure that n is not a member of any cycle that would violate the acyclicity condition of any of the nodes in S, including n.

DEFINITION 9. *We say that a node n satisfies a strong acyclicity check in graph $G = \langle H, \rho, K\rangle$ with respect to a set S, and we write* $\mathsf{acycCheckAll}(n, G, S)$, *iff it is not the case that: H contains a cycle $n_1, f_1, \ldots, n_s, f_s, n_1$ where $n_1 = n$, $f_1, \ldots, f_s \in \mathsf{acyclic}(\rho(n_i))$, for some $1 \le i \le s$, and $n_1, \ldots, n_s \in S$.*

acycCheckAll is a stronger condition than acycCheck because it ensures the absence of cycles containing n for $\mathsf{acyclic}(\rho(n_i))$ fields of all offstage nodes n_i, and not only for the fields $\mathsf{acyclic}(\rho(n))$.

The analysis uses the predicate nodeCheck to verify that bringing a node n offstage does not violate role consistency for offstage nodes.

DEFINITION 10. $\mathsf{nodeCheck}(n, G, S)$ *holds iff both predicates* $\mathsf{localCheck}(n, G)$ *and* $\mathsf{acycCheckAll}(n, G, S)$ *hold.*

6. INTERPROCEDURAL ROLE ANALYSIS

Our interprocedural role analysis can be seen as similar in the spirit to the functional approach to interprocedural dataflow analysis [42]. However, simply tagging a dataflow fact G with the abstract values of the initial procedure context G_0 is not appropriate for a complex abstraction such as role graphs. We instead approximate the transfer functions in the concrete semantics with procedure interfaces consisting of: 1) an initial context and 2) a set of *effects*. Effects summarize store statements and can naturally describe local heap modifications. We assume that procedure interfaces are supplied and we are concerned with a) verifying that procedure interfaces conservatively approximate behavior of the procedure, and b) instantiating procedure interfaces at the call sites.

6.1 Procedure Interfaces

A procedure interface for a procedure proc extends the procedure signature with an initial context $\mathsf{context}(\mathsf{proc})$, and procedure effects $\mathsf{effect}(\mathsf{proc})$.

6.1.1 Initial Context

An initial context is a description of the initial role graph of the procedure. The initial role graph specifies a set of concrete heaps at procedure entry and assigns names for the sets of objects in these heaps. We denote the initial role graph by $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$. The set of legal states at procedure entry is the set of concrete heaps that are represented by the initial role graph:

DEFINITION 11. *We say that a concrete heap $\langle H_c, \rho_c\rangle$ is represented by the initial role graph $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ and we write $\langle H_c, \rho_c\rangle\, \alpha_0 \langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$, iff there exists a function $h_0 : \mathsf{nodes}(H_c) \to \mathsf{nodes}(H_{\mathsf{IC}})$ such that*

1. $\mathsf{conW}(\rho_c, H_c, h_0^{-1}(\mathsf{read}(\mathsf{proc}))$;
2. *h_0 is a graph homomorphism;*
3. *$K_{\mathsf{IC}}(n) = i$ implies $|h_0^{-1}(n)| \le 1$;*
4. *$h_0(\mathsf{null}_c) = \mathsf{null}$ and $h_0(\mathsf{proc}_c) = \mathsf{proc}$;*
5. *$\rho_c(o) = \rho(h_0(o))$ for every object $o \in \mathsf{nodes}(H_c)$.*

Here $\mathsf{read}(\mathsf{proc})$ is the set of initial role graph nodes that represent the objects read by the procedure as specified by its read effects (Section *6.1.2*). For simplicity, we assume one context per procedure; it is straightforward to generalize the treatment to multiple contexts.

We explain the syntax of the initial role graph through the example of the `insert` procedure in Figure 10. The `insert` procedure inserts an isolated object with no slots or fields onto the end of an acyclic singly linked list. As a result, the role of the inserted object changes to `ListNode`.

Syntactically, each procedure specifies ρ_{IC} and K_{IC} using a `nodes` declaration and specifies H_{IC} using an `edges` declaration. The declaration `nodes` introduces individual and summary nodes and specifies a role for every node at procedure entry. Individual nodes are denoted with lowercase identifiers, summary nodes with uppercase identifiers. The procedure interface in Figure 10 introduces two individual nodes: `ln`, denoting the object referenced by the `l` parameter, and `xn`, denoting the object referenced by the `x` parameter. The roles of these nodes at procedure entry are `List` and `Isolated`, as the declaration of parameters `l` and `x` indicates. The interface can use individual and summary

nodes to specify the heap structure of objects that are not directly referenced by parameters. In this case the interface must explicitly specify the roles of these nodes. In general, the interface can use multiple summary nodes to specify the disjointness of heap regions and reachability properties.

The initial role graph may contain two kinds of edges: parameter edges and heap edges. A parameter edge `p->pn` is interpreted as $\langle \mathsf{proc}, \mathtt{p}, \mathtt{pn} \rangle \in H_{\mathsf{IC}}$. The role of a parameter node referenced by $\mathsf{param}_i(\mathsf{proc})$ is always $\mathsf{preR}_i(\mathsf{proc})$. A heap edge `n -f-> m` denotes an edge $\langle \mathtt{n}, \mathtt{f}, \mathtt{m} \rangle \in H_{\mathsf{IC}}$. The procedure interface in Figure 10 introduces two parameter edges `l->xn` and `x->xn`, and two heap edges `ln -next-> ListNode` and `ln -next-> null`. The two heap edges are written using a shorthand notation that groups edges with the same source.

We assume that the initial role graph always contains the role reference diagram RRD. We call nodes from the RRD *anonymous nodes*. The initial context descriptions denote these nodes with role names. The procedure interface in Figure 10 uses `ListNode` as an anonymous node. The description of the initial role graph specifies the edges between explicitly declared nodes as well as the edges between anonymous nodes and the explicitly declared nodes. In our example there is one edge, `ln -next-> ListNode`, from a declared node (`ln`) to an anonymous node (`ListNode`).

The analysis automatically derives the edges between the anonymous nodes from the RRD, leveraging the global role definitions to reduce the size of the initial role graph. A procedure can even entirely omit the initial context specification and use the RRD as a default initial role graph, as in Figure 3. As a notational convenience, the procedure interface description may use parameter names to denote the nodes referenced by the parameters.

6.1.2 *Procedure Effects*

Procedure effects conservatively approximate the region of the heap that the procedure accesses and indicate changes to the referencing relationships within this region. There are two kinds of effects: read effects and write effects.

A *read effect* specifies a set $\mathsf{read}(\mathsf{proc})$ of initial role graph nodes accessed by the procedure. It is used to ensure that the accessibility condition in Section *4.2.3* is satisfied. If the set of nodes $\mathsf{read}(\mathsf{proc})$ is mapped to a node n which is onstage in the caller but is not an argument of the procedure call, a role check error is reported at the call site. This check guarantees that the callee never accesses objects with temporarily violated roles.

A *write effect* of the form $e_1.f = e_2$ summarizes the effect of Store operations within the callee. The expression e_1 denotes the written objects, f denotes the written field, and e_2 denotes the objects whose references are written into the field. Write effects are *may* effects by default, which means that the procedure is free not to perform them. If the `!` keyword precedes the effect, the procedure *must* perform the effect.

In Figure 10 two write effects summarize the body of the procedure `insert`. The first write effect indicates that the procedure may perform zero or more Store operations to the `next` fields of objects mapped to `ln` or `ListNode` in $\mathsf{context}(\mathsf{proc})$. The second write effect indicates that the execution of the procedure must perform a Store to the `next` field of the object mapped to the `xn` node, where the reference stored is a `ListNode` object or `null`. A procedure

```
role List {
  fields next : ListNode | null;
  slots none;
}
role ListNode {
  fields next : ListNode | null;
  slots List.next | ListNode.next;
}
role Isolated { }
procedure insert(l: List,
                 x: Isolated ->> ListNode)
nodes ln, xn;
edges l-> ln, x-> xn,
      ln -next-> ListNode|null;
effects ln|ListNode . next = xn,
        ! (xn.next = ListNode|null);
var c, p;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  p.next = x;
  x.next = c;
  setRole(x:ListNode);
}
```

Figure 10: Insert Procedure for Acyclic List

interface may omit a read effect for a node that appears on the left-hand side of some write effect in the interface. Hence there is no need to declare read effects in Figure 3 or Figure 10.

The analysis uses write effects to modify the caller's role graph to conservatively model any writes that an execution of the procedure could perform.

Effects also summarize assignments that procedures perform to newly created objects. Here we adopt the simple solution of using a single summary node denoted NEW to represent all objects created inside the procedure. We write $\mathsf{nodes}_0(H_{\mathsf{IC}})$ for the set $\mathsf{nodes}(H_{\mathsf{IC}}) \cup \{\mathsf{NEW}\}$.

We represent all may write effects as a set $\mathsf{mayWr}(\mathsf{proc})$ of triples $\langle n, f, n' \rangle$ where $n, n' \in \mathsf{nodes}_0(H_{\mathsf{IC}})$ and $f \in F$. We represent must write effects as a sequence $\mathsf{mustWr}_j(\mathsf{proc})$ of subsets of the set $K_{\mathsf{IC}}^{-1}(i) \times F \times \mathsf{nodes}_0(H_{\mathsf{IC}})$. Here $1 \leq j \leq \mathsf{mustNo}(\mathsf{proc})$ where $\mathsf{mustNo}(\mathsf{proc})$ is the number of must write effects of procedure proc.

To simplify the interpretation of the declared procedure effects in terms of concrete reads and writes we require that the union $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$ must be disjoint from the set $\mathsf{mayWr}(\mathsf{proc})$. We also require the nodes $n_1, \ldots, n_k$ in a must write effect $n_1 | \cdots | n_k.f = e_2$ to be individual nodes. This allows strong updates when instantiating effects (Section *6.3.2*).

6.2 Verifying Procedure Interfaces

In this section we show how the analysis makes sure that a procedure conforms to its specification, expressed as an initial context with a list of effects. To verify the effects, we first extend the analysis representation from Section 5.1. A non-error role graph is now a tuple $\langle H, \rho, K, \tau, E \rangle$ where:

$$[\![\texttt{entry}\bullet]\!] = \Big\{ G_k \,\big|\, G_0 \underset{n'_1,p_1}{\overset{n_1}{\longmapsto}} \cdots \underset{n'_k,p_k}{\overset{n_k}{\longmapsto}} G_k,\ G_0 = \langle H_0, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}, \mathsf{id}, \emptyset\rangle,\ \mathsf{id}(x) = x,\ \text{for } x \in \mathsf{nodes}(H_0),\ H_p = \{\langle \mathsf{proc}, p_i, n\rangle \mid \langle \mathsf{proc}, p_i, n\rangle \in H_{\mathsf{IC}}\},\ H_0 = H_{\mathsf{IC}} \setminus H_p,\ \forall i\, \langle \mathsf{proc}, p_i, n'_i\rangle \in H_p \Big\}$$

$$G \underset{n',p}{\overset{n_0}{\longmapsto}} \langle H_2 \cup \{\langle \mathsf{proc}, p, n_0\rangle\}, \rho_2, K_2, \tau_2, E_2\rangle \quad \text{where } G \underset{n'}{\overset{n_0}{\Uparrow}} G_1 \overset{n_0}{\|} \langle H_2, \rho_2, K_2, \tau_2, E_2\rangle$$

Figure 11: Role Graphs at Entry to $\mathsf{proc}(p_1, \ldots, p_k)$

1. $\tau : \mathsf{nodes}(H) \to \mathsf{nodes}_0(H_{\mathsf{IC}})$ is an initial context transformation that assigns an initial role graph node $\tau(n) \in \mathsf{nodes}(H_{\mathsf{IC}})$ to every node n representing an object that existed prior to the procedure call. It also assigns NEW to every node representing an object created during the execution of the procedure;
2. $E \subseteq \cup_i \mathsf{mustWr}_i(\mathsf{proc})$ is a list of must write effects that the procedure has performed so far.

The initial context transformation τ tracks how objects have moved since the beginning of the procedure execution and is essential for the verification of the effects, which refer to initial role graph nodes.

We represent the list E of performed must effects as a partial map from the set $K_{\mathsf{IC}}^{-1}(i) \times F$ to $\mathsf{nodes}_0(H_{\mathsf{IC}})$. This representation allows the analysis to perform must effect "folding" by recording only the last must effect for every pair $\langle n, f\rangle$ of individual node n and field f.

6.2.1 Role Graphs at Procedure Entry

Our role analysis creates the set of role graphs at procedure entry from the initial role graph $\mathsf{context}(\mathsf{proc})$. This is conceptually simple because of the similarity of the abstraction relation for role graphs (Section 5.1) and the semantics of initial role graphs (Section 6.1). The difference is that in the role graph, the activation record proc has exactly one reference for each parameter, and this reference points to an instantiated onstage node. The analysis therefore instantiates the nodes in the initial role graph to create a set of role graphs that satisfy the representation invariant from Section 5.1.

Figure 11 describes this process. The analysis first selects one parameter edge for each parameter. It then uses the parameter expansion relation $\longmapsto$ to instantiate and split the referenced node. The parameter expansion relation $\longmapsto$ is similar to the expansion relation $\preceq$ from Section *5.2.1*.

6.2.2 Verifying Basic Statements

To ensure that a procedure conforms to its interface, the analysis uses the map τ to assign every Load and Store statement to a declared effect. Figure 12 describes this effect verification. The symbolic execution of a Load statement `x=y.f` makes sure that the load is recorded in some read effect. If this is not the case, an error is reported. The symbolic execution of the Store statement `x.f=y` first retrieves nodes $\tau(n_x)$ and $\tau(n_y)$ of the initial role graph that correspond to nodes n_x and n_y in the current role graph. If the effect $\langle \tau(n_x), f, \tau(n_y)\rangle$ is declared as a may write effect, the execution proceeds as usual. Otherwise, the effect is used to update the list E of must-write effects. The list E is checked at the end of procedure execution to allow the folding of the effects performed on the same node. The symbolic execution of the New statement updates the initial role graph transformation τ assigning $\tau(n_n) = \mathsf{NEW}$ for the new node n_n.

It is straightforward to lift the expansion and contraction relations from Section 5 to operate on role graphs containing τ and E as components. The expansion and contraction relations do not modify the list of performed effects E. Instantiation $\Uparrow$ of node n' into node n_0 assigns $\tau(n_0) = \tau(n')$, and splitting copies the values of τ into the nodes generated as a result of split. The normalization operation does not merge nodes n_1 and n_2 if $\tau(n_1) \neq \tau(n_2)$.

6.2.3 Verifying Procedure Postconditions

At the end of the procedure, the analysis verifies that $\rho(n_i) = \mathsf{postR}_i(\mathsf{proc})$ where $\langle \mathsf{proc}, \mathsf{param}_i(\mathsf{proc}), n_i\rangle \in H$, and then performs a node check on all onstage nodes using the predicate $\mathsf{nodeCheck}(n, \langle H, \rho, K\rangle, \mathsf{nodes}(H))$ for all nodes $n \in \mathsf{onstage}(H)$.

The analysis also verifies that every performed effect in E can be attributed to exactly one declared must effect. This means that for $k = \mathsf{mustNo}(\mathsf{proc})$ there exists a permutation s of the set $\{1, \ldots, k\}$ such that $e_{s(i)} \in \mathsf{mustWr}_i(\mathsf{proc})$ for all i, $1 \leq i \leq k$.

6.3 Analyzing Call Sites

The analysis updates the set of role graphs at the procedure call site based on the callee's interface. Consider a procedure proc with a call site of the form $\mathsf{proc}'(x_1, \ldots, x_p)$. Let $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ be the initial role graph of the callee proc'. The computation of the transfer function for a call site has the following phases:

1. **Parameter Check** ensures that roles of parameters conform to the roles expected by the callee proc'.
2. **Context Matching** (matchContext) ensures that the caller's role graphs represent a subset of the concrete heaps represented by $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ by deriving a mapping μ from the caller's role graph to $\mathsf{nodes}(H_{\mathsf{IC}})$.
3. **Effect Instantiation** ($\overset{\mathsf{FX}}{\longrightarrow}$) uses $\mathsf{mayWr}(\mathsf{proc}')$ and $\mathsf{mustWr}_i(\mathsf{proc}')$ to approximate all changes to the role graph that proc' may perform.
4. **Role Reconstruction** ($\overset{\mathsf{RR}}{\longrightarrow}$) uses final roles for parameter nodes, $\mathsf{postR}_i(\mathsf{proc}')$, combined with global role definitions, to reconstruct roles for all nodes in the part of the role graph representing modified region of the heap.

The parameter check requires $\mathsf{nodeCheck}(n_i, G, \mathsf{offstage}(H) \cup \{n_j\}_j)$ for every parameter node n_i. We next describe the other three phases.

6.3.1 Context Matching

Figure 13 presents our context matching function. The matchContext function takes a set $\mathcal{G}$ of role graphs and produces a set of pairs $\langle G, \mu\rangle$ where $G = \langle H, \rho, K, \tau, E\rangle$ is a role graph and μ is a homomorphism from H to H_{IC}. The homomorphism μ guarantees that $\alpha^{-1}(G) \subseteq \alpha_0^{-1}(\mathsf{context}(\mathsf{proc}'))$

Statement s	Transition	Constraints
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_f\}, \rho, K, \tau, E\rangle$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$ $\tau(n_f) \in \mathsf{read}(\mathsf{proc})$
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\Longrightarrow} \bot_G$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$ $\tau(n_f) \notin \mathsf{read}(\mathsf{proc})$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K, \tau, E\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \in \mathsf{mayWr}(\mathsf{proc})$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K, \tau, E'\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \in \cup_i \mathsf{mustWr}_i(\mathsf{proc})$ $E' = \mathsf{updateWr}(E, \langle \tau(n_x), f, \tau(n_y)\rangle)$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\Longrightarrow} \bot_G$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \notin \mathsf{mayWr}(\mathsf{proc}) \cup$ $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$
`x = new`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_n\}, \rho, K, \tau', E\rangle$	n_n fresh $\tau' = \tau[n_n \mapsto \mathsf{NEW}]$

$$\mathsf{updateWr}(E, \langle n_1, f, n_2\rangle) = E[\langle n_1, f\rangle \mapsto n_2]$$

Figure 12: Verifying that Load, Store, and New Statements Respect the Declared Effects

because the homomorphism h_0 from Definition 11 can be constructed from the homomorphism h in Definition 6 by putting $h_0 = \mu \circ h$. This implies that it is legal to call proc' with any concrete graph represented by G.

The algorithm in Figure 13 starts with empty maps $\mu = \mathsf{nodes}(G) \times \{\bot\}$ for each role graph G in the set of role graphs. The algorithm extends each map μ until μ is defined on all $\mathsf{nodes}(G)$ or there is no way to extend it further. It proceeds by choosing a role graph $\langle H, \rho, K, \tau, E\rangle$ and node n_0 for which the mapping μ is not defined yet. It then finds candidates in the initial role graph that n_0 can be mapped to. The accessibility requirement—that a procedure may access no object with an incorrect role—is enforced by making sure that nodes in $\mathsf{inaccessible}$ are never mapped onto nodes in read for the callee. As long as this requirement holds, nodes in $\mathsf{inaccessible}$ can be mapped onto nodes of any role since their role need not be correct anyway.

We generally require that the set $\mu^{-1}(n_0')$ for individual node n_0' in the initial role graph contain at most one node, and this node must be individual. In contrast, there might be many individual and summary nodes mapped onto a summary node. We relax this requirement and instantiate a summary node in the caller if, at some point, that is the only way to extend the mapping μ (this corresponds to the first recursive call in the definition of match in Figure 13).

The algorithm is nondeterministic in the order in which nodes to be matched are selected. One possible ordering of nodes is the depth-first order in the role graph starting from parameter nodes. If some nondeterministic branch does not succeed, the algorithm backtracks. The function fails if all branches fail. In that case the procedure call is considered illegal and $\bot_G$ is returned. The algorithm terminates since every recursive call to $\mathsf{matchContext}$ defines $\mu(n)$ where $\mu(n)$ was previously undefined.

6.3.2 *Effect Instantiation*

The result of the matching algorithm is a set of pairs $\langle G, \mu\rangle$ of role graphs and mappings. These pairs are used to instantiate the procedure effects in each of the role graphs of the caller. The analysis verifies that the region read by the callee is included in the region read by the caller. Then it uses μ to find the inverse image S of the performed write effects. The effect instantiation groups effects in S by the source n and field f. There are three cases when the analysis applies an effect to a node n and field f:

1. There is only one node in $\mathsf{nodes}(H)$ that is a target of the write effect, and the effect is a must write effect. In this case the analysis performs a strong update, which is possible because n is an individual node.
2. The condition in 1) is not satisfied, and the node n is offstage. In this case the analysis conservatively adds all relevant edges from S to H, using the fact that edges for offstage nodes are may edges.
3. The condition in 1) is not satisfied, but the node n is onstage i.e. it is a parameter node. In this case the analysis does a case analysis, choosing which write was performed last and generating a new role graph for each case. If there are no must effects that affect n, the analysis also generates the role graph in which the original references are unchanged.

6.3.3 *Role Reconstruction*

Our procedure effects approximate structural changes to the heap, but they do not provide information about role changes for non-parameter nodes. The role reconstruction algorithm $\overset{\mathsf{RR}}{\longrightarrow}$ in Figure 14 uses the role changes for the parameters and the global role definitions to conservatively infer possible roles for these nodes after the procedure call.

The role reconstruction algorithm first finds the set N_0 of all nodes that might be accessed by the callee (these are nodes whose roles may change). It then splits each node $n \in N_0$ into $|R|$ different nodes $\rho(n, r)$, one for each role $r \in R$. The node $\rho(n, r)$ represents the subset of objects that were initially represented by n and have role r after the procedure executes. The analysis derives edges between nodes in the new graph by simultaneously satisfying 1) structural constraints between nodes of the original graph; and 2) global role constraints from the role reference diagram. In this way, the analysis preserves the context-specific information on reachability, while assigning consistent roles to

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{RR}} \langle H',\rho',K',\tau',E'\rangle$$
$$H' = \mathsf{GC}(H_0), \qquad \langle \mathsf{proc}_j, x_i, n_i\rangle \in H, \qquad N_0 = \mu^{-1}[\mathsf{read}(\mathsf{proc}')]$$
$s : N_0 \times R \to N$ where $s(n,r)$ are all different nodes fresh in H
$$\rho' = \rho \setminus (N_0 \times R) \cup \{\langle s(n,r), r\rangle \mid n \in N_0, r \in R\} \setminus (\{n_i\}_i \times R) \cup \{\langle n_i, \mathsf{postR}_i(\mathsf{proc})\rangle\}$$
$$K'(s(n,r)) = K(n), \qquad \tau'(s(n,r)) = \tau(n), \qquad E' = E$$
$$\begin{aligned} H_0 = H \setminus \{\langle n_1, f, n_2\rangle \mid n_1 \in N_0 \text{ or } n_2 \in N_0\} \\ \cup \{\langle s(n_1,r_1), f, s(n_2,r_2)\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle r_1,f,r_2\rangle \in \mathsf{RRD}\} \\ \cup \{\langle n_1, f, s(n_2,r_2)\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle \rho_{\mathsf{IC}}(\mu(n_1)),f,r_2\rangle \in \mathsf{RRD}\} \\ \cup \{\langle s(n_1,r_1), f, n_2\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle r_1,f,\rho_{\mathsf{IC}}(\mu(n_2))\rangle \in \mathsf{RRD}\} \end{aligned}$$

Figure 14: Call Site Role Reconstruction

$$\mathsf{matchContext}(\mathcal{G}) = \mathsf{match}(\{\langle G, \mathsf{nodes}(G) \times \{\bot\}\rangle \mid G \in \mathcal{G}\})$$

$$\begin{aligned} \mathsf{match}: \quad & \mathcal{P}(\mathsf{RoleGraphs} \times (N \cup \{\bot\})^N) \\ \rightharpoonup \quad & \mathcal{P}(\mathsf{RoleGraphs} \times N^N) \end{aligned}$$

$\mathsf{match}(\Gamma) =$
$\Gamma_0 := \{\langle G,\mu\rangle \in \Gamma \mid \mu^{-1}(\bot) \neq \emptyset\}$;
if $\Gamma_0 = \emptyset$ then return Γ;
$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle :=$ choose Γ_0;
$\Gamma' = \Gamma \setminus \langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle$;
paramnodes $:= \{n \mid \exists i : \langle \mathsf{proc}, x_i, n\rangle \in H\}$;
inaccessible $:=$ onstage$(H) \setminus$ paramnodes;
$n_0 :=$ choose $\mu^{-1}(\bot)$;
candidates $:= \{n' \in \mathsf{nodes}(H_{\mathsf{IC}}) \mid$
$(n_0 \notin$ inaccessible and $\rho_{\mathsf{IC}}(n') = \rho(n_0))$ or
$(n_0 \in$ inaccessible and $n' \notin \mathsf{read}(\mathsf{proc}'))\}$
$$\bigcap_{\substack{\langle n_0,f,n\rangle \in H \\ \mu(n)\neq\bot}} \left\{n' \;\middle|\; \langle n', f, \mu(n)\rangle \in H_{\mathsf{IC}}\right\}$$
$$\bigcap_{\substack{\langle n,f,n_0\rangle \in H \\ \mu(n)\neq\bot}} \left\{n' \;\middle|\; \langle \mu(n), f, n'\rangle \in H_{\mathsf{IC}}\right\};$$
if candidates $= \emptyset$ then fail ;
if candidates $= \{n_0'\}, K(n_0) = s, K_{\mathsf{IC}}(n_0') = i, \mu^{-1}(n_0') = \emptyset$
then $\mathsf{match}(\Gamma' \cup \{\langle G', \mu[n_1 \mapsto n_0']\rangle \mid \langle H,\rho,K,\tau,E\rangle \overset{n_1}{\Uparrow}_{n_0} G'\})$
else $n_0' :=$ choose $\{n' \in$ candidates $\mid K(n') = s$ or
$(K(n_0) = i, \mu^{-1}(n') = \emptyset)\}$
$\mathsf{match}(\Gamma' \cup \langle\langle H,\rho,K,\tau,E\rangle,\mu[n_0 \mapsto n_0']\rangle)$;

Figure 13: The Context Matching Algorithm

all offstage nodes. The nodes $\rho(n,r)$ not connected to the parameter nodes are garbage collected in the role graph. In practice, we can generate nodes $\rho(n,r)$ on demand starting from the parameters, making sure that they are reachable while satisfying both kinds of constraints.

7. EXTENSIONS

This section presents extensions of the basic role system.

7.1 Cascading Role Changes

In some cases it is desirable to change the roles of an entire set of offstage objects without bringing them onstage. We use the statement `setRoleCascade`$(x_1 : r_1, \ldots, x_n : r_n)$ to perform a *cascading role change* on a set of objects. The need for cascading role changes arises when roles encode reachability properties.

Given a role graph $\langle H,\rho,K,E\rangle$, a cascading role change finds a new valid role assignment ρ' in which the onstage objects have the desired roles and the roles of offstage objects are adjusted appropriately. This operation makes sure it is legal to change the role of a node n from $\rho(n)$ to $\rho'(n)$ given that the neighbors of n also change role according to ρ'. This check resembles the check in the `setRole` statement of Section *5.2.3.2*.

There may be zero or more solutions that satisfy the constraints for a given cascading role change. Any solution that satisfies the constraints is sound with respect to the original semantics.

7.2 Simultaneous Roles

We next sketch an extension of our role framework to allow objects to play multiple roles simultaneously. In the context of simultaneous roles, each role specifies constraints on some of the fields and aliases of an object. An object playing multiple roles simultaneously satisfies the conjunction of the constraints associated with each role. For example, consider the definition of a tree:

```
role TreeHeader {
  fields left : TreeNode | null,
         right : TreeNode | null;
  left,right slots none;
}
role TreeNode {
  fields left : TreeNode | null,
         right : TreeNode | null;
  left,right slots : TreeHeader.left
                   | TreeHeader.right
                   | TreeNode.left
                   | TreeNode.right;
}
```

This definition specifies that a data structure is a tree along the `left` and `right` fields, but does not constrain fields other than `left` and `right`. Similarly, the definition of a linked list specifies constraints only for the `next` field:

```
role ListHeader {
  fields next : ListNode | null;
  next slots none;
}
role ListNode {
  fields next : ListNode | null;
  next slots ListHeader.next | ListNode.next;
}
```

Note how the definition of `ListHeader` specifies the absence of any aliases along the `next` field. We can combine the tree and list roles to obtain a threaded tree role:

```
role LinkedTreeNode extends TreeNode,ListNode { }
```

Every object playing the `LinkedTreeNode` role simultaneously plays both the `TreeNode` and `ListNode` roles.

In the context of simultaneous roles, not mentioning a field f in a definition of role r implies no constraints on the f-references of the objects playing role r. A slot constraint for a simultaneous role r contains an additional set $\mathsf{scope}(r) = \{f_1, \ldots, f_k\}$ of fields that determines the scope of the slot constraints. A slot declaration gives complete aliases for references from other objects along $\mathsf{scope}(r)$ fields, but imposes no constraints on aliases from other fields.

Role definitions for simultaneous roles can reuse previous role definitions via the `extends` keyword. We represent the `extends` relationships by the set of roles $\mathsf{subroles}(r)$ for each role r. A set $S \subseteq R$ is *closed* if $\mathsf{subroles}(r) \subseteq S$ for every $r \in S$.

To give semantics for simultaneous roles, we define a role-set assignment ρ_c^s to assign a closed set of roles to every object. We say that a role assignment ρ_c is a *choice* of a role-set assignment ρ_c^s iff $\rho_c(r) \in \rho_c^s(r)$ for every role $r \in R$. We first generalize $\mathsf{locallyConsistent}$ to take the role of the object o independently of the role assignment ρ_c. The following definition is similar to Definition 2.

DEFINITION 12. $\mathsf{locallyConsistent}(o, H_c, \rho_c, r)$ *iff all of the following conditions are met.*

1) *For every field* $f \in F$ *such that* field_f *is defined, if* $\langle o, f, o' \rangle \in H_c$, *then* $\rho_c(o') \in \mathsf{field}_f(r)$.
2) *Let* $\{\langle o_1, f_1 \rangle, \ldots, \langle o_k, f_k \rangle\} = \{\langle o', f \rangle \mid \langle o', f, o \rangle \in H_c, f \in \mathsf{scope}(r)\}$ *be the set of all* $\mathsf{scope}(r)$*-aliases of node* o. *Then* $k = \mathsf{slotno}(r)$ *and there exists some permutation* p *of the set* $\{1, \ldots, k\}$ *such that* $\langle \rho_c(o_i), f_i \rangle \in \mathsf{slot}_{p_i}(r)$ *for all* i.
3) *If* $\langle o, f, o' \rangle \in H_c$, $\langle o', g, o'' \rangle \in H_c$, *and* $\langle f, g \rangle \in \mathsf{identities}(r)$, *then* $o = o''$.
4) H_c *does not contain a cycle* $o_1, f_1, \ldots, o_s, f_s, o_1$ *where* $o_1 = o$ *and* $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$

We now define local role-set consistency as follows.

DEFINITION 13. $\mathsf{locallyRSConsistent}(o, H_c, \rho_c^s)$ *iff for each* $r \in \rho_c^s(o)$ *there exists a role assignment* ρ_c *which is a choice of* ρ_c^s *such that* $\mathsf{locallyConsistent}(o, H_c, \rho_c, r)$ *holds. We say that a heap* H_c *is role-set consistent for a role-set assignment* ρ_c^s *if* $\mathsf{locallyRSConsistent}(o, H_c, \rho_c^s)$ *for every* $o \in \mathsf{nodes}(H_c)$. *We call such role-set assignment* ρ_c^s *a valid role-set assignment.*

We similarly extend the definitions of consistency for a given set of objects from Definition 3.

Simultaneous roles enable us to develop a role system that supports a form of polymorphism. We allow role graphs at procedure call sites to have more specific roles than roles in the initial context of the callee. For example, a procedure for manipulating a linked list can be called on a node of a threaded tree. The analysis uses the procedure effects to identify the unmodified sets of references. Because the linked list operation does not modify the tree references, our analysis preserves the information about the tree edges across the procedure call.

8. RELATED WORK

Typestate, as a type system extension for statically verifying dynamically changing object properties, was originally proposed in [45, 44]. In this system, the state of an object depends only on its initialization status. In general, aliasing causes problems for typestate-based systems because the declared typestates of all aliases must change whenever the state of the referred object changes. Because of this problem, the original typestate system did not support references to dynamically allocated objects. More recently proposed typestate approaches use linear types for heap references to support state changes of dynamically allocated objects [10]. Researchers have also developed typestate approaches for verifying safety properties of assembly language programs [51, 52].

Motivated by the need to enforce safety properties in low-level software systems, several researchers have developed systems that use extensions of linear types to avoid general aliasing and rely on language design to avoid non-local type inference [43, 49, 9]. These systems take a *construction-based approach* that specifies data structures as unfoldings of basic elaboration steps [49]. Similarly to shape types [13] this approach can capture very precise properties of tree-like data structures but cannot approximate data structures such as sparse matrices.

Graph types and the Pointer Assertion Logic [28, 29, 26, 34] use monadic second-order logic [46] to describe heap invariants. In these systems, each data structure must be represented as a spanning tree backbone with additional pointer fields constrained to denote exactly one target node [34]. If a data structure is expressible in this way, the system can verify strong properties about it. Because of the constraints on pointer fields, however, it is not possible to approximate the full range of data structures. It is also impossible to describe objects that participate in multiple data structures at the same time because the system does not allow objects to be part of multiple backbones simultaneously.

Like shape analysis techniques [7, 15, 40, 41] we have adopted a *constraint-based approach* that characterizes data structures in terms of the constraints that they satisfy. This approach supports a wide range of data structures, but gives up some precision to achieve this generality. A high-level difference between previous approaches and our approach is that our approach focuses on global aspects such as the participation of objects in multiple data structures, while previous approaches focused on detailed properties of individual data structures. We do, however, use techniques similar to those developed previously to analyze detailed properties of a single data structure. For example, our instantiation relation is analogous to the materialization operation of [40, 41, 33], and our split operation [32, 31] achieves a similar goal to the focus operation of [41]. Like the parametric analysis of [41], our analysis uses reachability properties to keep nodes separate in the abstract graph, but differs in that it does not maintain an explicit reachability predicate for every node.

Researchers have developed a precise interprocedural analysis that treats activation records as dynamically allocated structures [37]. This approach also effectively synthesizes an application-specific set of contexts, eliminating any need for the program to specify additional interface information. Our approach differs in that it uses a less precise but more scalable treatment of procedures. It also uses a compositional approach that analyzes each procedure once in isolation to

verify that it conforms to its interface. The goal is to obtain an interface general enough to be used in any context, not just those contexts that happen to appear in the specific program at hand.

Context-sensitive pointer analyses [50, 17, 38] typically compute points-to relationships by caching generated contexts and using a fixpoint computation inside strongly connected components of the call graph. Our analysis tracks much more detailed information about the heap and uses procedure effects to achieve compositionality at the level of procedures.

The path matrix approaches [16, 15] have been used to implement efficient interprocedural analyses that infer one level of referencing relationships, but are not sufficiently precise to track must aliases of heap objects for programs with destructive updates of more complex data structures.

The ADDS and ASAP languages allow programmers to express data structure properties [24, 25, 21]. This work uses data structure invariants to enable compiler optimizations, but, unlike our role analysis, does not verify that operations on data structures preserve these invariants [23].

Several researchers have developed annotation languages to enable the separate analysis of multiple procedures, specifically for pointer analysis, bounds analysis of array indices, and effect analysis [19, 39]. The techniques presented in this paper track much more detailed aliasing properties. Our procedure effects are also more specific and precise than the effects in [27]; as a result they are not idempotent. Both verification and instantiation of our effects require specific techniques that precisely keep track of the correspondence between the initial heap of a procedure and the heap at each program point.

The object-oriented community has long been aware of the benefits of dynamically changing classes in large systems [36]. Recognizing these benefits, researchers have proposed dynamic techniques that change the class of an object to reflect its state changes [18, 6, 11, 14, 1, 53, 47]. These systems illustrate the need for a static system that can verify the correct use of objects with changing roles.

The presence of aliasing makes it difficult to enforce encapsulation in object-oriented languages. Motivated by this problem, researchers have developed several systems [22, 2, 8, 35] that restrict aliasing to prevent representation exposure. Many of these systems use unique references to limit aliasing, an approach that has also been proposed to verify the absence of data races in multithreaded programs [4] and to improve the ability of the compiler to generate optimized code [12]. The work on alias burying [5] shows how a static analysis can increase the flexibility of unique references. These approaches do not attempt to describe more general invariants of recursive data structures or participation of objects in data structures.

Model checking [3] can also be used to verify properties of dynamically allocated data structures, but additional techniques appear necessary to represent properties of heap regions not referenced directly by local variables.

9. CONCLUSION

This paper proposes two key ideas: aliasing relationships should determine, in part, the type of each object, and the type system should use the resulting object states as its fundamental abstraction for describing procedure interfaces and object referencing relationships. We present a role system that realizes these two key ideas and an analysis algorithm that can verify that the program correctly respects the role constraints. The result is that programmers can use roles for a variety of purposes: to ensure the correctness of extended procedure interfaces that take the roles of parameters into account, to verify important data structure consistency properties, to express how procedures move objects between data structures, and to check that the program correctly implements correlated relationships between the states of multiple objects. We therefore expect roles to improve the reliability of the program and its transparency to developers and maintainers.

10. REFERENCES

[1] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. *The VLDB Journal*, 4(3), 1995.

[2] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, 1997.

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[4] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 2001.

[5] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice & Experience*, 6(31):533–553, May 2001.

[6] Craig Chambers. Predicate classes. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 268–296, 1993.

[7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, 1990.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.

[9] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, 1999.

[10] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[11] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, LNCS 2072, pages 130–149. Springer, 2001.

[12] Jeffrey S. Foster and Alex Aiken. Checking programmer-specified non-aliasing. Technical Report CSD-01-1160, University of California, Berkeley, 2001.

[13] Pascal Fradet and Daniel Le Metayer. Shape types. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, 1997.

[14] Giorgio Ghelli and Debora Palmerini. Foundations for extensible objects with roles. In *Workshop on Foundations of Object-Oriented Languages, Paris*, 1999.

[15] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, 1996.

[16] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[17] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, 1998.

[18] Georg Gottlob, Michael Schrefl, and Brigitte Roeck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), 1994.

[19] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, 1999.

[20] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Mass., 2000.

[21] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[22] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991.

[23] Joseph Hummel. *Data Dependence Testing in the Presence of Pointers and Pointer-Based Data Structures*. PhD thesis, Dept. of Computer Science, Univ. of California at Irvine, 1998.

[24] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3), September 1993.

[25] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, Cancun, Mexico, April 26–29 1994.

[26] Jacob J. Jensen, Michael E. Joergensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, 1997.

[27] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Annual ACM Symposium on the Principles of Programming Languages*, 1991.

[28] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*, Charleston, SC, 1993.

[29] Nils Klarlund and Michael I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. 19th Colloquium on Trees and Algebra in Programming*, number 787 in LNCS, 1994.

[30] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, 1999.

[31] Viktor Kuncak. Designing an algorithm for role analysis. Master's thesis, Massachusetts Institute of Technology, 2001.

[32] Viktor Kuncak, Patrick Lam, and Martin Rinard. Roles are really great! Technical Report 822, Laboratory for Computer Science, Massachusetts Institute of Technology, `http://www.mit.edu/~vkuncak/papers/`, 2001.

[33] Tal Lev-Ami. TVLA: A framework for kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[34] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[35] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, 1998.

[36] Trygve Reenskaug. *Working With Objects*. Prentice Hall, 1996.

[37] Noam Rinetzky and Mooly Sagiv. Interprocedual shape analysis for recursive programs. In *Proceedings of the 10th International Conference on Compiler Construction*, 2001.

[38] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[39] Radu Rugina and Martin Rinard. Design-driven compilation. In *Proceedings of the 10th International Conference on Compiler Construction*, 2001.

[40] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, 1996.

[41] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, 1999.

[42] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.

[43] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the 14th European Symposium on Programming*, Berlin, Germany, March 2000.

[44] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, May 1993.

[45] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.

[46] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.

[47] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[48] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, 1990.

[49] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.

[50] R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[51] Zhichen Xu, Barton Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000.

[52] Zhichen Xu, Thomas Reps, and Barton Miller. Typestate checking of machine code. In *Proceedings of the 15th European Symposium on Programming*, 2001.

[53] Phillip M. Yelland. Experimental classification facilities for Smalltalk. In *Proceedings of the 6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1992.