Roles Are Really Great!

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 for specifying changing referencing relationships of heap objects. The role of an object depends, in large part, on its aliasing relationships with other objects, with the role of each object changing as its aliasing relationships change. Roles therefore 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 respects the role constraints.

Contents

*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.

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 [44, 43, 10, 47, 48, 4, 20, 13]. These systems integrate the concept of changing object states into the type system.

The fundamental idea in this paper is that the state of each object also depends 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 [45, 2, 30] by allowing multiple aliases to be statically tracked, and extend alias types [42, 46] with the ability to specify 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 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.
- **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 specifications. The algorithm uses a data-flow analysis to infer intermediate referencing relationships between objects, allowing the programmer to focus on role changes and procedure interfaces.

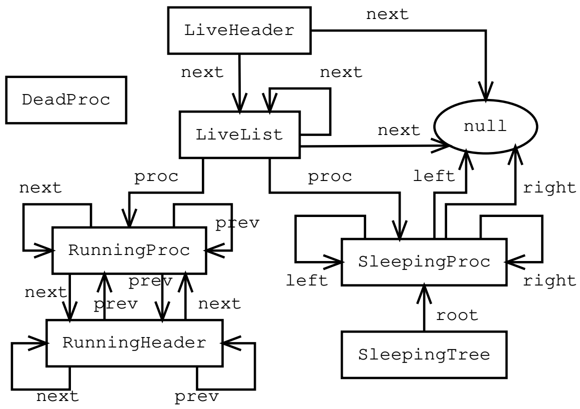


Figure 1: Role Reference Diagram for Scheduler

2 Example

Figure 1 presents a *role reference diagram* for a process scheduler. Each box in the diagram denotes a disjoint set of objects of 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 sleeping processes form a binary tree. Both kinds of processes have `proc` references from the live list nodes `LiveList`. Header objects `RunningHeader` and `SleepingTree` simplify operations on the data structures that store the process objects.

As Figure 1 shows, data structure participation determines the conceptual state of each object. In our example, processes that participate in the sleeping process tree data structure are classified as sleeping processes, while processes that participate in the running process list data structure 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 for the objects in our example.[1] Each role definition specifies the constraints that an object must satisfy to play the role. Field constraints

[1] In general, each role definition would specify the static class of objects that can play that role. To simplify the

specify the roles of the objects to which the fields refer, while slot constraints identify the number and kind of aliases of the object.

```
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 SleepingTree {
  fields root : SleepingProc | null,
  acyclic left, right;
}
role SleepingProc {
  fields left  : SleepingProc | null,
         right : SleepingProc | null;
  slots SleepingProc.left | SleepingProc.right |
        SleepingTree.root;
        LiveList.proc;
  acyclic left, right;
}
role DeadProc {  }
```

Figure 2: Role Definitions for a Scheduler

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`.

The slot constraints specify the complete set of heap aliases for the object. In our example, this implies that no process can be simultaneously running and sleeping.

presentation, we assume that all objects are instances of a single class with a set of fields F.

In general, roles can capture data structure consistency properties such as disjointness and can prevent representation exposure [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: an object with role `RunningProc p`, and the `SleepingTree s`. The procedure changes the role of the object referenced by `p` to `SleepingProc` whereas the object referenced by `s` retains its original role. To perform the role change, the procedure removes `p` from its `RunningList` data structure and inserts it into the `SleepingTree` data structure `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.

```
procedure suspend(p : RunningProc ->> SleepingProc,
                  s : SleepingTree)
local 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

3 Abstract 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 a 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 objects of 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. We do not consider subtyping or dynamic dispatch in this paper.

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 let

$R_0 = R \cup \{\mathsf{null}_R\}$. 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 a slot k of object o if and only if $\langle r', f\rangle \in \mathsf{slot}_k(r)$. An object with role r must have each of its slots filled by exactly one reference.

- **Identities:** Every role $r \in R$ has a set of $\mathsf{identities}(r) \subseteq F \times F$. 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.

3.3 Role Semantics

We define the semantics of roles as a conjunction of invariants associated with 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 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. Let $r = \rho_c(o)$.*

1) *For every field $f \in F$ and $\langle o, f, o'\rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.*
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 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) *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 $o_1 \neq o_2$. This is just one of the ways in which roles make aliasing more predictable.

4 Role Properties

Roles capture important properties of the objects and provide useful information about how the actions of the program affect those properties.

- **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.

- **Multiple Uses:** Code factoring minimizes code duplication by producing general-purpose classes (such as the Java Vector and Hashtable classes) that can be used in a variety of contexts. But this practice obscures the different purposes that different instances of these classes serve in the computation. Because each instance's purpose is usually reflected in its relationships with other objects, roles can often recapture these distinctions.

- **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.

Programmers can use roles for specifying the membership of objects in data structures and the structural invariants of data structures. In both cases, the slot constraints are essential.

When used to describe membership of an object in a data structure, slots specify the source of the alias from a data structure node that stores the object. By assigning different sets of roles to data structures used at different program points, it is possible to distinguish nodes stored in different data structure instances. As an object moves between data structures, the role of the object changes appropriately to reflect the new source of the alias.

When describing nodes of data structures, slot constraints specify the aliasing constraints of nodes; this is enough to precisely describe a variety of data structures and approximate many others. Property 16 below shows how to identify trees in role definitions even if tree nodes have additional aliases from other sets of nodes. It is also possible to define nodes which make up a compound data structure linked via disjoint sets of fields, such as threaded trees, sparse matrices and skip lists.

Example 3 The following role definitions specify a sparse matrix of width and height at least 3. These definitions can be easily constructed from a sketch of a sparse matrix, as in Figure 4.

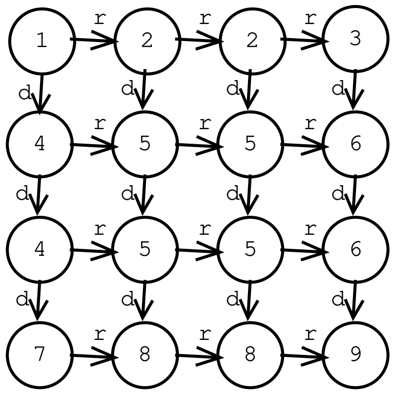


Figure 4: Roles of Nodes of a Sparse Matrix

```
role A1 {
  fields right : A2, down : A4;
  acyclic right, down;
}
role A2 {
  fields right : A2 | A3, down : A5;
  slots A1.right | A2.right;
  acyclic right, down;
}
role A3 {
  fields down : A6;
  slots A2.right;
  acyclic right, down;
}
role A4 {
  fields right : A5, down : A4 | A7;
  slots A1.down | A4.down;
  acyclic right, down;
}
role A5 {
  fields right : A5 | A6, down : A5 | A8;
  slots A4.right | A5.right, A2.down | A5.down;
  acyclic right, down;
}
role A6 {
  fields down : A6 | A9;
  slots A5.right, A3.down | A6.down;
  acyclic right, down;
}
role A7 {
  fields right : A8;
  slots A4.down;
  acyclic right, down;
}
role A8 {
  fields right : A8 | A9;
  slots A7.right | A8.right, A5.down;
  acyclic right, down;
}
role A9 {
  slots A8.right, A6.down;
  acyclic right, down;
```

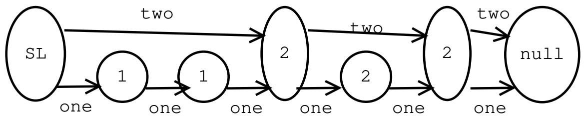


Figure 5: Sketch of a Two-Level Skip List

```
}
```

Example 4 We next give role definitions for a two-level skip list [36] sketched in Figure 5.

```
role SkipList {
  fields one : OneNode | TwoNode | null;
         two : TwoNode | null;
}
role OneNode {
  fields one : OneNode | TwoNode | null;
         two : null;
  slots OneNode.one | TwoNode.one | SkipList.one;
  acyclic one, two;
}
role TwoNode {
  fields one : OneNode | TwoNode | null;
         two : TwoNode | null;
  slots OneNode.one | TwoNode.one | SkipList.one,
        TwoNode.two | SkipList.two;
  acyclic one, two;
}
```

4.1 Formal Properties of Roles

In this section we identify some of the invariants expressible using sets of mutually recursive role definitions. A further study of role properties can be found in [31].

The following properties show some of the ways role specifications make object aliasing more predictable. They are an immediate consequence of the semantics of roles.

Property 5 *(Role Disjointness)*
If there exists a valid role assignment ρ_c *for* H_c *such that* $\rho(o_1) \neq \rho(o_2)$, *then* $o_1 \neq o_2$.

The previous property gives a simple criterion for showing that objects o_1 and o_2 are unaliased: find a valid role assignment which assigns different roles to o_1 and o_2. This use of roles generalizes the use of static types for pointer analysis [12]. Since roles create a finer partition of objects than a typical static type system, their potential for proving absence of aliasing is even larger.

Property 6 *(Disjointness Propagation)*
If $\langle o_1, f, o_2\rangle, \langle o_3, g, o_4\rangle \in H_c$, $o_1 \neq o_3$, *and there exists a valid role assignment* ρ_c *for* H_c *such that* $\rho_c(o_2) = \rho_c(o_4) = r$ *but* $\mathsf{field}_f(r) \cap \mathsf{field}_g(r) = \emptyset$, *then* $o_2 \neq o_4$.

Property 7 *(Generalized Uniqueness)*
If $\langle o_1, f, o_2\rangle, \langle o_3, g, o_4\rangle \in H_c$, $o_1 \neq o_3$, *and there exists a role assignment* ρ_c *such that* $\rho_c(o_2) = \rho_c(o_4) = r$, *but there are no indices* $i \neq j$ *such that* $\langle \rho_c(o_1), f\rangle \in \mathsf{slot}_i(r)$ *and* $\langle \rho_c(o_2), g\rangle \in \mathsf{slot}_j(r)$ *then* $o_2 \neq o_4$.

A special case of Property 7 occurs when $\mathsf{slotno}(r) = 1$; this constrains all references to objects of role r to be unique.

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

Definition 8 *(Role Reference Diagram)*
Given a set of definitions of roles R, a role reference diagram RRD is is a directed graph with nodes R_0 and labelled edges defined by

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

Each role reference diagram is a refinement of the corresponding class diagram in a statically typed language, because it partitions classes into multiple roles according to their referencing relationships. The sets $\rho_c^{-1}(r)$ of objects with role r change during program execution, reflecting the changing referencing relationships of objects.

Role definitions give more information than a role reference diagram. Slot constraints specify not only that objects of role r_1 can reference objects of role r_2 along field f, but also give cardinalities on the number of references from other objects. In addition, role definitions include identity and acyclicity constraints, which are not present in role reference diagrams.

Property 9 *Let ρ_c be any valid role assignment. Define*

$$G = \{\langle \rho_c(o_1), f, \rho_c(o_2)\rangle \mid \langle o_1, f, o_2\rangle \in H_c\}$$

Then G is a subgraph of RRD.

It follows from Property 9 that roles give an approximation of may-reachability among heap objects.

Property 10 *(May Reachability)*
If there is a valid role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\rho_c(o_1) \neq \rho_c(o_2)$ where $o_1, o_2 \in \mathsf{nodes}(H_c)$ and there is no path from $\rho_c(o_1)$ to $\rho_c(o_2)$ in the role reference diagram RRD, then there is no path from o_1 to o_2 in H_c.

The next property shows the advantage of explicitly specifying null references in role definitions. While the ability to specify acyclicity is provided by the `acyclic` constraint, it is also possible to indirectly specify must-cyclicity.

Property 11 *(Must Cyclicity)*
Let $F_0 \subseteq F$ and $R_{\mathsf{CYC}} \subseteq R$ be a set of nodes in the role reference diagram RRD such that for every node $r \in R_{\mathsf{CYC}}$, if $\langle r, f, r'\rangle \in \mathsf{RRD}$ then $r' \in R_{\mathsf{CYC}}$. If ρ_c is a valid role assignment for H_c, then every object $o_1 \in H_c$ with $\rho_c(o_1) \in R_{\mathsf{CYC}}$ is a member of a cycle in H_c with edges from F_0.

The following property shows that roles can specify a form of must-reachability among the sets of objects with the same role.

Property 12 *(Downstream Path Termination)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{FINAL}} \subseteq R_0$ of the role reference diagram RRD such that for every node $r \in R_{\mathsf{INTER}}$:

1. *$F_0 \subseteq \mathsf{acyclic}(r)$*
2. *if $\langle r, f, r'\rangle \in \mathsf{RRD}$ for $f \in F_0$, then $r' \in R_{\mathsf{INTER}} \cup R_{\mathsf{FINAL}}$*

Let ρ_c be a valid role assignment for H_c. Then every path in H_c starting from an object o_1 with role $\rho_c(o_1) \in R_{\mathsf{INTER}}$ and containing only edges labelled with F_0 is a prefix of a path that terminates at some object o_2 with $\rho_c(o_2) \in R_{\mathsf{FINAL}}$.

Property 13 *(Upstream Path Termination)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{INIT}} \subseteq R_0$ of the role reference diagram RRD such that for every node $r \in R_{\mathsf{INTER}}$:

1. *$F_0 \subseteq \mathsf{acyclic}(r)$*
2. *if $\langle r', f, r\rangle \in \mathsf{RRD}$ for $f \in F_0$, then $r' \in R_{\mathsf{INTER}} \cup R_{\mathsf{INIT}}$*

Let ρ_c be a valid role assignment for H_c. Then every path in H_c terminating at an object o_2 with $\rho_c(o_2) \in R_{\mathsf{INTER}}$ and containing only edges labelled with F_0 is a suffix of a path which started at some object o_1, where $\rho_c(o_1) \in R_{\mathsf{INIT}}$.

The next two properties guarantee reachability properties by which there must exist at least one path in the heap, rather than stating properties of all paths as in Properties 12 and 13.

Property 14 *(Downstream Must Reachability)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of roles $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{FINAL}} \subset R_0$ of the role reference diagram RRD such that for every node $r \in R_{\mathsf{INTER}}$:

1. *$F_0 \subseteq \mathsf{acyclic}(r)$*
2. *there exists $f \in F_0$ such that $\mathsf{field}_f(r) \subseteq R_{\mathsf{INTER}} \cup R_{\mathsf{FINAL}}$*

Let ρ_c be a valid role assignment for H_c. Then for every object o_1 with $\rho_c(o_1) \in R_{\mathsf{INTER}}$ there is a path in H_c with edges from F_0 from o_1 to some object o_2 where $\rho_c(o_2) \in R_{\mathsf{FINAL}}$.

Property 15 *(Upstream Must Reachability)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{INIT}} \subseteq R$ of the role reference diagram RRD such that for every node $r \in R_{\mathsf{INTER}}$:

1. *$F_0 \subseteq \mathsf{acyclic}(r)$*
2. *there exists k such that $\mathsf{slot}_k(r) \subseteq (R_{\mathsf{INTER}} \cup R_{\mathsf{INIT}}) \times F$*

Let ρ_c be a valid role assignment for H_c. Then for every object o_2 with $\rho_c(o_2) \in R_{\mathsf{INTER}}$ there is a path in H_c from some object o_1 with $\rho_c(o_1) \in R_{\mathsf{INIT}}$ to the object o_2.

Trees are a class of data structures especially suited for static analysis. Roles can express graphs that are not trees, but it is useful to identify trees as certain sets of mutually recursive role definitions.

Property 16 *(Treeness)*
Let $R_{\mathsf{TREE}} \subseteq R$ be a set of roles and $F_0 \subseteq F$ set of fields such that for every $r \in R_{\mathsf{TREE}}$

1. *$F_0 \subseteq \mathsf{acyclic}(r)$*
2. *$|\{i \mid \mathsf{slot}_i(r) \cap (R_{\mathsf{TREE}} \times F_0) \neq \emptyset\}| \leq 1$*

Let ρ_c be a valid role assignment for H_c and $S \subseteq \{\langle n_1, f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in H_c, \rho(n_1), \rho(n_2) \in R_{\mathsf{TREE}}, f \in F_0\}$. Then S is a set of trees.

5 A Programming Model

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

We first present a simple imperative language with dynamic object allocation and give its operational semantics. We then specify additional statement preconditions that enforce the role consistency requirements.

5.1 A Simple Imperative Language

Our core language contains, as basic statements, Load (`x=y.f`), Store (`x.f=y`), Copy (`x=y`), and New (`x=new`). All variables are references to objects in the global heap and all assignments are reference assignments. We use an elementary `test` statement combined with nondeterministic choice and iteration to express `if` and `while` statement, using the usual translation [22, 1]. 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 $\mathsf{local}(\mathsf{proc})$. We use $\mathsf{var}(\mathsf{proc})$ to denote $\mathsf{param}(\mathsf{proc}) \cup \mathsf{local}(\mathsf{proc})$. 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 activation records of procedure proc. We further assume that there are no modifications of parameter variables so every parameter references the same object throughout the lifetime of procedure activation.

Example 17 The following `kill` procedure removes a process from both the doubly linked list of running processes and the list of all active processes. This is indicated by the transition from `RunningProc` to `DeadProc`.

```
procedure kill(p : RunningProc ->> DeadProc,
               l : LiveHeader)
local prev, current, cp, nxt, lp, ln;
{
  // find 'p' in 'l'
  prev = l; current = l.next;
  cp = current.proc;
  while (cp != p) {
    prev = current;
    current = current.next;
    cp = current.proc;
  }
  // remove 'current' and 'p' from active list
  nxt = current.next;
  prev.next = nxt; current.
  current.proc = null;
  setRole(current : IsolatedCell);
  // remove 'p' from running list
  lp = p.prev;   ln = p.next;
  p.prev = null; p.next = null;
  lp.next = ln;   ln.prev = lp;
  setRole(p : DeadProc);
}
```

5.2 Operational Semantics

In this section we give the operational semantics for our language. We focus on the first three columns in Figures 6 and 7; the safety conditions in the fourth column are detailed in Section 5.4.

Figure 6 gives the small-step operational semantics for the basic statements. We use $A \uplus B$ to denote the union $A \cup B$ where the sets A and B are disjoint. The program state consists of the stack s and the concrete heap H_c. The stack s is a sequence of pairs $p@\mathsf{proc}_i \in \times(\mathsf{Proc} \times \mathcal{N})$, where $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ is a program point, and $\mathsf{proc}_i \in \mathsf{Proc} \times \mathcal{N}$ is an activation record of procedure proc. Program points $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ are nodes of the control-flow graphs. There is one control-flow graph for every procedure proc. An edge of the control-flow graph $\langle p, p' \rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$ indicates that control may transfer from point p to point p'. We write $p : \mathsf{stat}$ to state that program point p contains a statement stat. The control flow graph of each procedure contains special program points `entry` and `exit` indicating procedure entry and exit, with no statements associated with them. We assume that all conditions are of the form `x==y` or `!(x==y)` where `x` and `y` are either variables or a special constant `null` which always points to the null_c object.

The concrete heap is either an error heap error_c or a non-error heap. A non-error heap $H_c \subseteq N \times F \times N \cup ((\mathsf{Proc} \times \mathcal{N}) \times V \times N)$ is a directed graph with labelled edges, where nodes represent objects and procedure activation records, whereas edges represent heap references and local variables. An edge $\langle o_1, f, o_2 \rangle \in N \times F \times N$ denotes a reference from object o_1 to object o_2 via field $f \in F$. An edge $\langle \mathsf{proc}_i, \mathtt{x}, o \rangle \in H_c$ means that local variable `x` in activation record proc_i points to object o.

A load statement `x=y.f` makes the variable `x` point to node o_f, which is referenced by the `f` field of object o_y, which is in turn referenced by variable `y`. A store statement `x.f=y` replaces the reference along field `f` in object o_x by a reference to object o_y that is referenced by `y`. The copy statement `x=y` copies a reference to object o_y into variable `x`. The statement `x=new` creates a new object o_n with all fields initially referencing null_c, and makes `x` point to o_n. The statement `test(c)` allows execution to proceed only if condition `c` is satisfied.

Figure 7 describes the semantics of procedure calls. Procedure call pushes new activation record onto stack, inserts it into the heap, and initializes the parameters. Procedure entry initializes local variables. Procedure exit removes the activation record from the heap and the stack.

5.3 Onstage and Offstage Objects

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

Statement	Transition	Constraints	Role Consistency
p : `x=y.f`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c'\rangle$	$\mathtt{x},\mathtt{y} \in \mathsf{local}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle, \langle o_y, \mathtt{f}, o_f\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H_c' = H_c \uplus \{\mathsf{proc}_i, \mathtt{x}, o_f\}$	$\mathsf{accessible}(o_f, \mathsf{proc}_i, H_c)$, $\mathsf{con}(H_c', \mathsf{offstage}(H_c'))$
p : `x.f=y`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle o_x, f, o_f\rangle\}\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c'\rangle$	$\mathtt{x},\mathtt{y} \in \mathsf{local}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle, \langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H_c' = H_c \uplus \{\langle o_x, f, o_y\rangle\}$	$o_f \in \mathsf{onstage}(H_c, \mathsf{proc}_i)$ $\mathsf{con}(H_c', \mathsf{offstage}(H_c'))$
p : `x=y`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c'\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, $\mathtt{y} \in \mathsf{var}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H_c' = H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_y\rangle\}$	$\mathsf{con}(H_c', \mathsf{offstage}(H_c'))$
p : `x=new`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c'\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, o_n fresh, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H_c' = H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_n\rangle\} \uplus \mathsf{nulls}$, $\mathsf{nulls} = \{o_n\} \times F \times \{\mathsf{null}\}$	$\mathsf{con}(H_c', \mathsf{offstage}(H_c'))$
p : `test(c)`	$\langle p@\mathsf{proc}_i; s, H_c\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c\rangle$	$\mathsf{satisfied}_c(\mathsf{c}, \mathsf{proc}_i, H_c)$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$	$\mathsf{con}(H_c, \mathsf{offstage}(H_c))$

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

$$\mathsf{satisfied}_c(\mathtt{!(x==y)}, \mathsf{proc}_i, H_c) \text{ iff not } \mathsf{satisfied}_c(\mathtt{x==y}, \mathsf{proc}_i, H_c)$$

$$\begin{array}{lcl} \mathsf{accessible}(o, \mathsf{proc}_i, H_c) & := & (\exists p \in \mathsf{param}(\mathsf{proc}) : \langle \mathsf{proc}_i, p, o\rangle \in H_c) \\ & & \text{or not } (\exists \mathsf{proc}'_j\ \exists v \in \mathsf{var}(\mathsf{proc}') : \langle \mathsf{proc}'_j, v, o\rangle \in H_c) \end{array}$$

Figure 6: Semantics of Basic Statements

Statement	Transition	Constraints	Role Consistency
`entry` : _	$\langle p@\mathsf{proc}_i; s, H_c\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c \uplus \mathsf{nulls}\rangle$	$\mathsf{nulls} = \{\langle \mathsf{proc}_i, v, \mathsf{null}_c\rangle \mid v \in \mathsf{local}(\mathsf{proc})$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$	$\mathsf{con}(H_c, \mathsf{offstage}(H_c))$
$p : \mathsf{proc}'(x_k)_k$	$\langle p@\mathsf{proc}_i; s, H_c\rangle \longrightarrow$ $\langle \mathtt{entry}@\mathsf{proc}'_j; p'@\mathsf{proc}_i; s, H_c'\rangle$	j fresh in $p@\mathsf{proc}i; s$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $o_k : \langle \mathsf{proc}_i, x_k, o_k\rangle \in H_c$, $H_c' = H_c \uplus \{\langle \mathsf{proc}'_j, p_k, o_k\rangle\}_k$, $\forall k\ p_k = \mathsf{param}_k(\mathsf{proc}')$	$\mathsf{conW}(\mathsf{ra}, H_c, S)$, $\mathsf{ra} = \{\langle o_k, \mathsf{preR}_k(\mathsf{proc}')\rangle\}_k$, $S = \mathsf{offstage}(H_c) \cup \{o_k\}_k$
`exit` : _	$\langle p@\mathsf{proc}_i; s, H_c\rangle \longrightarrow$ $\langle s, H_c \setminus \mathsf{AF}\rangle$	$\mathsf{AF} = \{\langle \mathsf{proc}_i, v, n\rangle \mid \langle \mathsf{proc}_i, v, n\rangle \in H_c\}$	$\mathsf{conW}(\mathsf{ra}, H_c, S)$, $\mathsf{ra} = \{\langle \mathsf{parnd}_k(\mathsf{proc}_i), \mathsf{postR}_k(\mathsf{proc})\rangle\}_k$, $S = \mathsf{offstage}(H_c) \cup \{o \mid \langle \mathsf{proc}_i, v, o\rangle \in H_c\}$

$$\mathsf{parnd}_k(\mathsf{proc}_i) = o \text{ where } \langle \mathsf{proc}_i, \mathsf{param}_k(\mathsf{proc}), o\rangle \in H_c$$

Figure 7: Semantics of Procedure Call

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

$$\begin{aligned}\mathsf{onstage}(H_c, \mathsf{proc}_i) &:= \{o \mid \exists x \in \mathsf{var}(\mathsf{proc}) \\ &\qquad \langle \mathsf{proc}_i, x, o\rangle \in H_c\} \\ \mathsf{onstage}(H_c) &:= \bigcup_{\mathsf{proc}_i} \mathsf{onstage}(H_c, \mathsf{proc}_i)\end{aligned}$$

2. **offstage objects** ($\mathsf{offstage}(H_c)$) unreferenced by local or parameter variables.

$$\mathsf{offstage}(H_c) := \mathsf{nodes}(H_c) \setminus \mathsf{onstage}(H_c)$$

Onstage objects need not have correct roles. Offstage objects must have correct roles assuming some role assignment for onstage objects; the exception is that acyclicity constraints for offstage objects can be violated due to cycles that pass through the onstage objects.

Definition 18 *Given a set of role definitions and a set of objects $S_c \subseteq \mathsf{nodes}(S_c)$, we say that 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 every object $o \in S_c$.*

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 19 $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ *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; 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 with 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 20 *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.*

5.4 Role Consistency

We are now able to precisely state the role consistency requirements that must be satisfied for program execution. The role consistency requirements are in the fourth row of Figures 6 and 7. We assume the operational semantics is extended with transitions leading to a program state with heap error_c whenever role consistency is violated.

5.4.1 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.

5.4.2 Reference Removal Consistency

The Store statement `x.f=y` has the following safety precondition. When 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 $\langle o_x, \mathtt{f}, o_f\rangle \in H_c$ is removed from the heap, both o_x and o_f must be referenced from the current procedure activation record. It is sufficient to verify this condition for o_f, as 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.

5.4.3 Procedure Call Consistency

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

A procedure call $\mathsf{proc}'(x_1, ..., x_p)$ in Figure 7 requires the role consistency precondition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, where the partial role assignment ra requires objects o_k, corresponding to parameters x_k, to have roles $\mathsf{preR}_k(\mathsf{proc}')$ expected by the callee, and $S_c = \mathsf{offstage}(H_c) \cup \{o_k\}_k$ for $\langle \mathsf{proc}_j, x_k, o_k\rangle \in H_c$.

To ensure that the callee proc'_j never observes incorrect roles, we impose an *accessibility condition* for the callee's Load statements (see the fourth column of Figure 6). The accessibility condition prohibits access to any object o referenced by some local variable of a stack frame 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. In Section 7 we show how the role analysis ensures that the accessibility condition is never violated.

At the procedure exit point (Figure 7), we require correct roles for all objects referenced by the current activation frame proc'_j. This implies that heap operations performed by proc'_j preserve heap consistency for all objects accessed by proc'_j.

5.4.4 Explicit Role Check

The programmer can specify a stronger invariant at any program point using statement $\mathtt{roleCheck}(x_1, \ldots, x_p, \mathsf{ra})$. As Figure 8 indicates, `roleCheck` requires the $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$ predicate to be satisfied for the supplied partial role assignment ra where $S_c = \mathsf{offstage}(H_c) \cup \{o_k\}_k$ for objects o_k referenced by given local variables x_k.

Statement	Transition	Constraints	Role Consistency
$p:\texttt{roleCheck}(x_1,\ldots,x_n,\mathsf{ra})$	$\langle p@\mathsf{proc}_i;s,H_c\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i;s,H_c\rangle$	$\langle p,p'\rangle \in E_{\mathsf{CFG}}$	$\mathsf{conW}(\mathsf{ra},H_c,S),$ $S = \mathsf{offstage}(H_c) \cup \{o \mid \langle \mathsf{proc}_i, x_k, o\rangle \in H_c\}$

Figure 8: **Operational Semantics of Explicit Role Check**

5.5 Instrumented Semantics

We expect the programmer to have a specific role assignment in mind when writing the program, with this role assignment changing as the statements of the program change the referencing relationships. So 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 ρ_c is conceptually part of the program's state. The role assignment changes only if the programmer changes it explicitly using the `setRole` statement. The augmented programming model has an underlying *instrumented semantics* as opposed to the *original semantics*.

Example 21 The original semantics allows asserting different roles at different program points even if the structure of the heap was not changed, as in the following procedure `foo`.

```
role A1 { fields f : B1; }
role B1 { slots A1.f; }
role A2 { fields f : B2; }
role B2 { slots A2.f; }
procedure foo()
var x, y;
{
  x = new;  y = new;
  x.f = y;
  roleCheck(x,y, x:A1,y:B1);
  roleCheck(x,y, x:A2,y:B2);
}
```

Both role checks would succeed since each of the specified partial role assignments can be extended to a valid role assignment. On the other hand, the check `roleCheck(x,y, x:A1,y:B2)` would fail.

The procedure `foo` in the instrumented semantics can be written as folllows.

```
procedure foo()
var x, y;
{
  x = new;  y = new;
  x.f = y;
  setRole(x:A1);  setRole(y:B1);
  roleCheck(x,y, x:A1,y:B1);
  setRole(x:A2);  setRole(y:B2);
  roleCheck(x,y, x:A2,y:B2);
}
```

The `setRole` statement makes the role change of object explicit.

The instrumented semantics extends the concrete heap H_c with a role assignment ρ_c. Figure 9 outlines the changes in instrumented semantics with respect to the original semantics. 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. Here we consider $\mathsf{con}(H_c,S)$ to be a shorthand for $\mathsf{conW}(\emptyset,H_c,S)$. For example, the new role consistency condition for the Copy statement `x=y` is $\mathsf{conW}(\rho_c,H_c,\mathsf{offstage}(H_c))$. The New statement assigns an identifier $\mathsf{unknown}$ to the newly created object o_n. By definition, a node with $\mathsf{unknown}$ does not satisfy the $\mathsf{locallyConsistent}$ predicate. This means that `setRole` must be used to set a a valid role of o_n before o_n moves offstage.

By introducing an instrumented semantics we are not suggesting an implementation that explicitly stores roles of objects at run-time. 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.

6 Intraprocedural Role Analysis

This section presents an intraprocedural role analysis algorithm. The goal of the role analysis is to statically verify the role consistency requirements described in the previous section.

The key observation behind our analysis algorithm is that we can incrementally verify role consistency of the concrete heap H_c 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 from and to onstage nodes. The exact representation of onstage nodes allows the analysis to verify role consistency in the presence of temporary violations of role constraints.

Our analysis representation is a graph in which nodes represent objects and edges represent references between ob-

[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 8.2.

Statement	Transition	Constraints	Role Consistency
p : x=new	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}, \rho_c\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H'_c, \rho'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, o_n fresh, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c$ $\uplus\{\langle \mathsf{proc}_i, \mathtt{x}, o_n\rangle\}$ $\uplus\{o_n\} \times F \times \{\mathsf{null}\}$, $\rho'_c = \rho_c[o_n \mapsto \mathsf{unknown}]$	$\mathsf{conW}(\rho'_c, H'_c, \mathsf{offstage}(H'_c))$
p : setRole(x:r)	$\langle p@\mathsf{proc}_i; s, H_c, \rho_c\rangle \longrightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c, \rho'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc}_i)$, $\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle \in H_c$, $\rho'_c = \rho_c[o_x \mapsto \mathtt{r}]$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}$	$\mathsf{conW}(\rho'_c, H_c, \mathsf{offstage}(H_c))$
p : stat	$\langle s, H_c, \rho_c\rangle \longrightarrow$ $\langle s', H'_c, \rho_c\rangle$	$\langle s, H_c\rangle \longrightarrow \langle s', H'_c\rangle$	$P \wedge \mathsf{conW}(\rho_c \cup \mathsf{ra}, H''_c, S)$ for every original condition $P \wedge \mathsf{conW}(\mathsf{ra}, H''_c, S)$

Figure 9: Instrumented Semantics

jects. 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 corresponding to 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. Distinct offstage objects with the same role r represent disjoint sets of objects of role r with different reachability properties from onstage nodes.

We frame role analysis as a data-flow 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. In Section 7 we generalize the algorithm to the compositional interprocedural analysis.

6.1 Abstraction Relation

Every data-flow 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 $\perp_G$ 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 $x \in \mathsf{local}(\mathsf{proc})$. Null references are represented as references to abstract node null. We define abstract onstage nodes $\mathsf{onstage}(H) = \{n \mid \langle \mathsf{proc}, x, n\rangle \in H, x \in \mathsf{local}(\mathsf{proc}) \cup \mathsf{param}(\mathsf{proc})\}$ and abstract 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 concrete heap and concrete role assignment with an abstract role graph G.

Definition 22 *We say that an abstract role graph G represents concrete heap H_c with role assignment ρ_c and write $\langle H_c, \rho_c\rangle \,\alpha\, G$, iff $G = \perp_G$ 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 $o \in \mathsf{nodes}(H_c)$.*

Note that the error heap error_c can be represented only by the bottom role graph $\perp_G$. The analysis uses $\perp_G$ 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 an edge in H from an onstage node

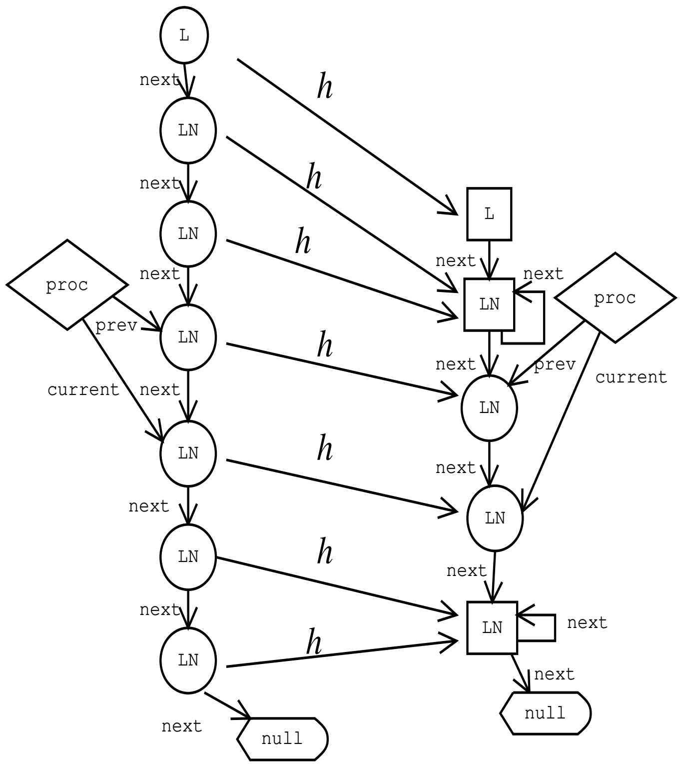


Figure 10: Abstraction Relation

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 6.2.4.

Example 23 Consider the following role declaration for an acyclic list.

```
role L { // List header
  fields first : LN | null;
}
role LN { // List node
  fields next : LN | null;
  slots LN.next | L.first;
  acyclic next;
}
```

Figure 10 shows a role graph and one of the concrete heaps represented by the role graph via homomorphism h. There are two local variables, `prev` and `current`, referencing distinct onstage objects. Onstage objects are isomorphic to onstage nodes in the role graph. In contrast, there are two objects mapped to each of the summary nodes with role `LN` (shown as `LN`-labelled rectangles in Figure 10). Note that the sets of objects mapped to these two summary nodes are disjoint. The first summary `LN`-node represents objects stored in the list before the object referenced by `prev`. The second summary `LN`-node represents objects stored in the list after the object referenced by `current`.

$$\begin{tikzcd}
 & & \langle H_c, \rho_c \rangle \arrow[dll, "\alpha"'] \arrow[d, "\alpha"] \arrow[rr] & & \langle H'_c, \rho'_c \rangle \arrow[drr, "\alpha"] & & \\
G_1 & \preceq & G_2 & \overset{st}{\Longrightarrow} & G_3 & \succeq & G_4
\end{tikzcd}$$

Figure 11: Simulation Relation Between Abstract and Concrete Execution

6.2 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 $\mathsf{locallyConsistent}$ 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. The expansion relation uses the invariant that offstage nodes have correct roles to generate possible aliasing relationships for the node being pulled onstage. The contraction relation establishes the role invariants for the node going offstage, allowing the node to be merged into the other offstage nodes and represented more compactly.

We present our role analysis as an abstract execution relation $\overset{\mathtt{st}}{\leadsto}$. The abstract execution ensures that the abstraction relation α is a forward simulation relation [33] from the space of concrete heaps with role assignments to the set $\mathsf{RoleGraphs}$. The simulation relation implies that the traces of $\leadsto$ include the traces of the instrumented semantics $\longrightarrow$. To ensure that the program does not violate constraints associated with roles, it is thus sufficient to guarantee that $\perp_G$ is not reachable via $\leadsto$.

To prove that $\perp_G$ 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 $\mathtt{st}$ is an image $[\![\mathtt{st}]\!](\mathcal{G}) = \{G' \mid G \in \mathcal{G}, G \overset{\mathtt{st}}{\leadsto} G'\}$. The analysis computes the relation $\overset{\mathtt{st}}{\leadsto}$ in three steps:

1. ensure that the relevant nodes are instantiated using expansion relation $\preceq$ (Section 6.2.1);

2. perform symbolic execution $\overset{\mathtt{st}}{\Longrightarrow}$ of the statement `st` (Section 6.2.3);

3. merge nodes if needed using contraction relation $\succeq$ to keep the role graph bounded (Section 6.2.2).

Figure 11 shows how the abstraction relation α relates $\preceq$, $\overset{\mathtt{st}}{\Longrightarrow}$, and $\succeq$ with the concrete execution $\longrightarrow$ in instrumented semantics. Assume that a concrete heap $\langle H_c, \rho_c \rangle$ is represented by the role graph G_1. Then one of the role graphs G_2 obtained after expansion remains an abstraction of $\langle H_c, \rho_c \rangle$.

Transition	Definition	Conditions
$\langle H,\rho,K\rangle \overset{\texttt{x=y.f}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{n_y,f}{\preceq} G_1 \overset{\texttt{x=y.f}}{\Longrightarrow} G_2 \overset{n_x}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$
$\langle H,\rho,K\rangle \overset{\texttt{x=y}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{\texttt{x=y}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_1\rangle \in H$
$\langle H,\rho,K\rangle \overset{\texttt{x=new}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{\texttt{x=new}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_1\rangle \in H$
$\langle H,\rho,K\rangle \overset{\texttt{s}}{\rightsquigarrow} G'$	$\langle H,\rho,K\rangle \overset{\texttt{s}}{\Longrightarrow} G'$	$s \in \{$`x.f=y`, `test(c)`, `setRole(x:r)`, `roleCheck(`$x_{1..p}$`, ra)`$\}$

Figure 12: Abstract Execution $\rightsquigarrow$

The symbolic execution $\overset{\texttt{st}}{\Longrightarrow}$ followed by the contraction relation $\succeq$ corresponds to the instrumented operational semantics $\rightarrow$.

Figure 12 shows rules for the abstract execution relation $\overset{\texttt{st}}{\rightsquigarrow}$. Only Load statement uses the expansion relation, because the other statements operate on objects that are already onstage. Load, Copy, and New statements may remove a local variable reference from an object, so they use contraction relation to move the object offstage if needed. For the rest of the statements, the abstract execution reduces to symbolic execution $\Longrightarrow$ described in Section 6.2.3.

Nondeterminism and Failure The $\overset{\texttt{st}}{\rightsquigarrow}$ relation is not a function because the expansion relation $\preceq$ can generate a set of role graphs from a single role graph. Also, there might be no $\overset{\texttt{st}}{\rightsquigarrow}$ transitions originating from a given state G if the symbolic execution $\Longrightarrow$ produces no results. This corresponds to a trace which cannot be extended further due to a `test` statement which fails in state G. This is in contrast to a transition from G to $\perp_G$ which indicates a potential role consistency violation or a null pointer dereference. We assume that $\Longrightarrow$ and $\succeq$ relations contain the transition $\langle \perp_G, \perp_G\rangle$ to propagate the error role graph. In most cases we do not write the explicit transitions to error states.

6.2.1 Expansion

Figure 13 shows the expansion relation $\overset{n,f}{\preceq}$. Given a role graph $\langle H,\rho,K\rangle$ expansion 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 $K(n_0) = i$. Expansion is used in abstract execution of the Load statement. It first checks for null pointer dereference and reports an error if the check fails. If $\langle n,f,n'\rangle \in H$ and $K(n') = i$ already hold, the expansion returns the original state. Otherwise, $\langle n,f,n'\rangle \in H$ with $K(n') = s$. In that case, the summary node n' is first instantiated using instantiation relation $\underset{n'}{\overset{n_0}{\Uparrow}}$. Next, the split relation $\overset{n_0}{\|}$ is applied. Let $\rho(n_0) = r$. The split relation ensures that n_0 is not a member of any cycle of offstage nodes which contains only edges in $\mathsf{acyclic}(r)$. We explain instantiation and split in more detail below.

Instantiation Figure 14 presents the instantiation relation. Given a role graph $G = \langle H,\rho,K\rangle$, instantiation $\underset{n'}{\overset{n_0}{\Uparrow}}$ generates the set of role graphs $\langle H',\rho',K'\rangle$ such that each concrete heap represented by $\langle H,\rho,K\rangle$ is represented by one of the graphs $\langle H',\rho',K'\rangle$. Each of the new role graphs contains a fresh individual node n_0 that satisfies $\mathsf{localCheck}$. The edges of n_0 are a subset of edges from and to n'.

Let H_0 be a subset of the references between n' and onstage nodes, and let H_1 be a subset of the references between n' and offstage nodes. References in H_0 are moved from n' to the new node n_0, because they represent at most one reference, while references in H_1 are copied to n_0 because they may represent multiple concrete heap references. Moving a reference is formalized via the swing operation in Figure 14.

The instantiation of a single graph can generate multiple role graphs depending on the choice of H_0' and H_1'. The number of graphs generated is limited by the existing references of node n' and by the $\mathsf{localCheck}$ requirement for n_0. This is where our role analysis takes advantage of constraints associated with role definitions to reduce the number of aliasing possibilities that need to be considered.

Split The split relation is important for verifying operations on data structures such as skip lists and sparse matrices. It is also useful for improving the precision of the initial set of role graphs on procedure entry (Section 7.2.1).

The goal of the split relation is to exploit the acyclicity constraints associated with role definitions. After a node n_0 is brought onstage, split represents the acyclicity condition of $\rho(n_0)$ explicitly by eliminating impossible paths in the role graph. It uses additional offstage nodes to encode the reachability information implied by the acyclicity conditions. This information can then be used even after the role of node n_0 changes. In particular, it allows the acyclicity condition of n_0 to be verified when n_0 moves offstage.

Example 24 Consider a role graph for an acyclic list with nodes `LN` and a header node `L`. The instantiated node n_0 is in the middle of the list. Figure 16 a) shows a role graph with a single summary node representing all offstage `LN`-nodes. Figure 16 b) shows the role graph after applying the split relation. The resulting role graph contains two `LN` summary nodes. The first `LN` summary node represents objects definitely reachable from n_0 along `next` edges; the second summary `NL` node represents objects definitely not reachable from n_0.

Figure 15 shows the definition of the split operation on node n_0, denoted by $\overset{n_0}{\|}$. Let $G = \langle H,\rho,K\rangle$ be the initial

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

Figure 13: Expansion Relation

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

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

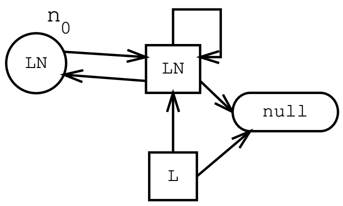
Figure 14: Instantiation Relation $\Uparrow$

$$\begin{array}{ll}\langle H,\rho,K\rangle \overset{n_0}{\parallel} \langle H,\rho,K\rangle, & \mathsf{acycCheck}(n_0, \langle H,\rho,K\rangle, \mathsf{offstage}(H)) \\ \langle H,\rho,K\rangle \overset{n_0}{\parallel} \langle H',\rho',K'\rangle, & \neg\mathsf{acycCheck}(n_0, \langle H,\rho,K\rangle, \mathsf{offstage}(H))\end{array}$$

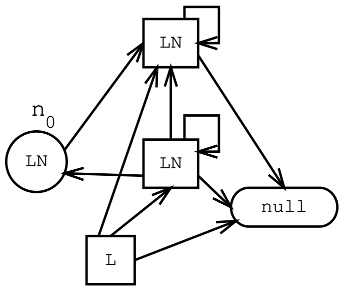
where

$$\begin{aligned}
& H' = (H \setminus H_{\text{cyc}}) \cup H_{\text{off}} \cup B_{\text{fNR}} \cup B_{\text{fR}} \cup B_{\text{tNR}} \cup B_{\text{tR}} \cup N_{\text{f}} \cup N_{\text{t}} \\
& H_{\text{cyc}} = \{\langle n_1, f, n_2\rangle \mid n_1 \text{ or } n_2 \in S_{\text{cyc}}\} \\
& H_{\text{off}} \quad = \quad \{\ \langle n_1', f, n_2'\rangle \quad \mid \quad n_1 = c(n_1'), n_2 = c(n_2'), \\
& \qquad\qquad\qquad\qquad\qquad n_1, n_2 \in \mathsf{offstage}_1(H), n_1 \text{ or } n_2 \in S_{\text{cyc}}, \\
& \qquad\qquad\qquad\qquad\qquad \langle n_1, f, n_2\rangle \in H\ \} \\
& \qquad\quad \setminus (S_{\text{R}} \times \mathsf{acyclic}(r) \times S_{\text{NR}}) \\
& H \cap (\mathsf{onstage}(H) \times F \cup \{n_0\} \times \mathsf{acyclic}(r)) \times S_{\text{cyc}} = A_{\text{fNR}} \uplus A_{\text{fR}} \\
& H \cap S_{\text{cyc}} \times (\mathsf{acyclic}(r) \times \{n_0\} \cup F \times \mathsf{onstage}(H)) = A_{\text{tNR}} \uplus A_{\text{tR}} \\
& B_{\text{fNR}} = \{\langle n_1, f, h_{\text{NR}}(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{fNR}}\} \\
& B_{\text{fR}} = \{\langle n_1, f, h_{\text{R}}(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{fR}}\} \\
& B_{\text{tNR}} = \{\langle h_{\text{NR}}(n_1), f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{tNR}}\} \\
& B_{\text{tR}} = \{\langle h_{\text{R}}(n_1), f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{tR}}\} \\
& N_{\text{f}} = \{\langle n_0, f, n'\rangle \mid n' \in S_{\text{R}}, \langle n_0, f, c(n')\rangle \in H, f \in \mathsf{acyclic}(r)\} \\
& N_{\text{t}} = \{\langle n', f, n_0\rangle \mid n' \in S_{\text{NR}}, \langle c(n'), f, n_0\rangle \in H, f \in \mathsf{acyclic}(r)\} \\
& S_{\text{cyc}} = \{n \mid \exists n_1, \ldots, n_{p-1} \in \mathsf{offstage}(H) : \\
& \qquad \langle n_0, f_0, n_1\rangle, \ldots, \langle n_k, f_k, n\rangle, \langle n, f_{k+1}, n_{k+2}\rangle, \langle n_{p-1}, f_{p-1}, n_0\rangle \in H, \\
& \qquad f_0, \ldots, f_{p-1} \in \mathsf{acyclic}(r)\} \\
& \mathsf{offstage}_1(H) = \mathsf{offstage}(H) \setminus \{n_0\} \\
& r = \rho(n_0) \\
& \\
& \rho'(c(n)) = \rho(n) \\
& K'(c(n)) = K(n)
\end{aligned}$$

Figure 15: Split Relation



a) Before Split



b) After Split

Figure 16: A Role Graph for an Acyclic List

role graph and $\rho(n_0) = r$. If $\mathsf{acyclic}(r) = \emptyset$, then the split operation returns the original graph G; otherwise it proceeds as follows. Call a path in graph H *cycle-inducing* if all of its nodes are offstage and all of its edges are in $\mathsf{acyclic}(r)$. Let S_{cyc} be the set of nodes n such that there is a cycle-inducing path from n_0 to n and a cycle-inducing path from n to n_0.

The goal of the split operation is to split the set S_{cyc} into a fresh set of nodes S_{NR} representing objects definitely not reachable from n_0 along edges in $\mathsf{acyclic}(r)$ and a fresh set of nodes S_{R} representing objects definitely reachable from n_0. Each of the newly generated graphs H' has the following properties:

1) merging the corresponding nodes from S_{NR} and S_{R} in H' yields the original graph H;
2) n_0 is not a member of any cycle in H' consisting of offstage nodes and edges in $\mathsf{acyclic}(r)$;
3) onstage nodes in H' have the same number of fields and aliases as in H.

Let $S_0 = \mathsf{nodes}(H) \setminus S_{\mathrm{cyc}}$ and let $h_{\mathrm{NR}} : S_{\mathrm{cyc}} \to S_{\mathrm{NR}}$ and $h_{\mathrm{R}} : S_{\mathrm{cyc}} \to S_{\mathrm{R}}$ be bijections. Define a function $c : \mathsf{nodes}(H') \to \mathsf{nodes}(H)$ as follows:

$$c(n) = \begin{cases} n, & n \in S_0 \\ h_{\mathrm{R}}^{-1}(n), & n \in S_{\mathrm{R}} \\ h_{\mathrm{NR}}^{-1}(n), & n \in S_{\mathrm{NR}} \end{cases}$$

Then $H' \subseteq \{\langle n_1', f, n_2' \rangle \mid \langle c(n_1'), f, c(n_2') \rangle \in H\}$.

Because there are two copies of S_0 in H', there might be multiple edges $\langle n_1', f, n_2' \rangle$ in H' corresponding to an edge $\langle c(n_1), f, c(n_2) \rangle \in H$.

If both n_1' and n_2' are offstage nodes other than n_0, we always include $\langle n_1', f, n_2' \rangle$ in H' unless $\langle n_1', f, n_2' \rangle \in S_{\mathrm{R}} \times \mathsf{acyclic}(r) \times S_{\mathrm{NR}}$. The last restriction prevents cycles in H'.

For an edge $\langle n_1, f, n_2 \rangle \in H$ where $n_1 \in \mathsf{onstage}(H)$ and $n_2 \in S_{\mathrm{cyc}}$ we include in H' either the edge $\langle n_1, f, h_{\mathrm{NR}}(n_2) \rangle$ or $\langle n_1, f, h_{\mathrm{R}}(n_2) \rangle$ but not both. Split generates multiple graphs H' to cover both cases. We proceed analogously if $n_2 \in \mathsf{onstage}(H)$ and $n_1 \in S_{\mathrm{cyc}}$. The node n_0 itself is treated in the same way as onstage nodes for $f \notin \mathsf{acyclic}(r)$. If $f \in \mathsf{acyclic}(r)$ then we choose references *to* n_0 to have a source in S_{NR}, whereas the reference *from* n_0 have the target in S_{R}.

Details of the split construction are given in Figure 15. The intuitive meaning of the sets of edges is the following:

H_{off} : edges between offstage nodes
B_{fNR} : edges from onstage nodes to S_{NR}
B_{fR} : edges from onstage nodes to S_{R}
B_{tNR} : edges from S_{NR} to onstage nodes
B_{tR} : edges from S_{R} to onstage nodes
N_{f} : $\mathsf{acyclic}(r)$-edges from n_0 to S_{R}
N_{t} : $\mathsf{acyclic}(r)$-edges from S_{NR} to n_0

The sets B_{fNR} and B_{fR} are created as images of the sets A_{fNR} and A_{fR} which partition edges from onstage nodes to nodes in S_{cyc}. Similarly, the sets B_{tNR} and B_{tR} are created as images of the sets A_{tNR} and A_{tR} which partition edges from nodes in S_{cyc} to onstage nodes.

We note that if in the split operation $S_{\mathrm{cyc}} = \emptyset$ then the operation has no effect and need not be performed. In Figure 16, after performing a single split, there is no need to split for subsequent elements of the list. Examples like this indicate that split will not be invoked frequently during the analysis.

6.2.2 Contraction

Figure 17 shows the non-error transitions of the contraction relation $\overset{n}{\succeq}$. The analysis uses contraction when a local variable reference to node n is removed. If there are other local references to n, the result is the original graph. Otherwise n has just gone offstage, so analysis invokes $\mathsf{nodeCheck}$. If the check fails, the result is $\bot_G$. If the role check succeeds, the contraction invokes normalization operation to ensure that the role graph remains bounded. For simplicity, we use normalization whenever $\mathsf{nodeCheck}$ succeeds, although it is sufficient to perform normalization only at program points adjacent to back edges of the control-flow graph.

Normalization Figure 18 shows the normalization relation. Normalization accepts a role graph $\langle H, \rho, K \rangle$ and produces a normalized role graph $\langle H', \rho', K' \rangle$ which is a factor graph of $\langle H, \rho, K \rangle$ under the equivalence relation $\sim$. Two offstage nodes are equivalent under $\sim$ if they have the same role and the same reachability from onstage nodes. Here we consider node n to be reachable from an onstage node n_0 iff there is some path from n_0 to n whose edges belong to $\mathsf{acyclic}(\rho(n_0))$ and whose nodes are all in $\mathsf{offstage}(H)$. Note that, by construction, normalization avoids merging nodes which were previously generated in the split operation $\|$, while still ensuring a bound on the size of the role graph. For a procedure with l local variables, f fields and r roles the number of nodes in a role graph is on the order of $r2^l$ so the

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

Figure 17: Contraction Relation

$$\mathsf{normalize}(\langle H, \rho, K\rangle) = \langle H', \rho', K'\rangle$$

where $H' = \{\langle n_{1/\sim}, f, n_{2/\sim}\rangle \mid \langle n_1, f, n_2\rangle \in H\}$
$\rho'(n_{/\sim}) = \rho(n)$
$K'(n_{/\sim}) = \begin{cases} i, & n_{/\sim} = \{n\}, K(n) = i \\ s, & \text{otherwise} \end{cases}$
$n_1 \sim n_2$ iff $n_1 = n_2$ or
$(n_1, n_2 \in \mathsf{offstage}(H), \rho(n_1) = \rho(n_2),$
$\forall n_0 \in \mathsf{onstage}(H) : (\mathsf{reach}(n_0, n_1)$ iff $\mathsf{reach}(n_0, n_2))$
$\mathsf{reach}(n_0, n)$ iff $\exists n_1, \ldots, n_{p-1} \in \mathsf{offstage}(n), \exists f_1, \ldots, f_p \in \mathsf{acyclic}(\rho(n_0)) :$
$\langle n_0, f_1, n_1\rangle, \ldots, \langle n_{p-1}, f_p, n\rangle \in H$

Figure 18: Normalization

maximum size of a chain in the lattice is of the order of 2^{r2^l}. To ensure termination we consider role graphs equal up to isomorphism. Isomorphism checking can be done efficiently if normalization assigns canonical names to the equivalence classes it creates.

6.2.3 Symbolic Execution

Figure 19 shows the symbolic execution relation $\overset{\mathtt{st}}{\Longrightarrow}$. In most cases, the symbolic execution of a statement 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 symbolic execution is due to conditions 3) and 5) in the abstraction relation α. These conditions are ensured by the $\preceq$ relation which instantiates nodes, allowing strong updates. The symbolic execution also verifies the consistency conditions that are not verified by $\preceq$ or $\succeq$.

Verifying Reference Removal Consistency The abstract execution $\overset{\mathtt{st}}{\leadsto}$ for the Store statement can easily verify the Store safety condition from section 5.4.2, because the set of onstage and offstage nodes is known precisely for every role graph. It returns $\bot_G$ if the safety condition fails.

Symbolic Execution of setRole The `setRole(x:r)` statement sets the role of node n_x referenced by 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 $h(o_x) = n_x$. Let $r_0 = \rho_c(o_x)$. 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 nodes 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 nodes, so it remains satisfied. The other role constraints are local, so they can only be violated for offstage neighbors of n_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 22 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.

Symbolic Execution of roleCheck To symbolically execute $\mathtt{roleCheck}(x_1, \ldots, x_p, \mathsf{ra})$, we ensure that the conW predicate of the concrete semantics is satisfied for the concrete heaps which correspond to the current abstract role graph. The symbolic execution for `roleCheck` returns the error graph $\bot_G$ if ρ is inconsistent with ra or if any of the nodes n_i referenced by x_i fail to satisfy nodeCheck.

6.2.4 Node Check

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 22.

Statement s	Transition	Conditions
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_f\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $n_f \in \mathsf{onstage}(H)$
`x = y`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_y\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$
`x = new`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_n\}, \rho', K\rangle$	n_n fresh $\rho' = \rho[n_n \mapsto \mathsf{unknown}]$
`test(c)`	$\langle H, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H, \rho, K\rangle$	$\mathsf{satisfied}(\mathsf{c}, H)$
`setRole(x:r)`	$\langle H, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H, \rho[n_x \mapsto \mathtt{r}], K\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle \in H$ $\mathsf{roleChOk}(n_x, \mathtt{r}, \langle H, \rho, K\rangle)$
`roleCheck(`$x_{1..p}$, ra`)`	$\langle H, \rho, K\rangle \overset{\mathsf{s}}{\Longrightarrow} \langle H, \rho, K\rangle$	$\forall i\ \langle \mathsf{proc}, \mathtt{x}_i, n_i\rangle \in H$ $\mathsf{nodeCheck}(n_i, \langle H, \rho, K\rangle, S)$ $S = \mathsf{offstage}(H) \cup \{n_i\}_i$ $\rho(n_i) = \mathsf{ra}(n_i)$

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

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

Figure 19: Symbolic Execution of Basic Statements

Definition 25 *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. Let $r = \rho(n)$.*

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 19 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 the Definition 22 to ensure that 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 19 for the node that has just been brought onstage. The split operation uses $\mathsf{acycCheck}$ to check whether it should split any nodes (Section 6.2.1). The set S represents offstage nodes.

Definition 26 *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 6.2.2) 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 27 *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 $1 \leq i \leq s$, and $n_1, \ldots, n_s \in S$.*

$\mathsf{acycCheckAll}$ is a stronger condition than $\mathsf{acycCheck}$ 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 $\mathsf{nodeCheck}$ to verify that bringing a node n offstage does not violate role consistency for offstage nodes.

Definition 28 $\mathsf{nodeCheck}(n, G, S)$ *holds iff both predicates* $\mathsf{localCheck}(n, G)$ *and* $\mathsf{acycCheckAll}(n, G, S)$ *hold.*

7 Interprocedural Role Analysis

This section describes the interprocedural aspects of our role analysis. Interprocedural role analysis can be viewed as an instance of the functional approach to interprocedural data-flow analysis [41]. For each program point p, role analysis approximates program traces from procedure entry to point p. The solution in [41] proposes tagging the entire data-flow fact G at point p with the data flow fact G_0 at procedure entry. In contrast, our analysis computes the correspondence between heaps at procedure entry and heaps at point p at the granularity of sets of objects that constitute role graphs. This allows our analysis to detect which regions of the heap have been modified. We approximate the concrete executions of a procedure with *procedure transfer relations* consisting of 1) an initial context and 2) a set of *effects*. Effects are fine-grained transfer relations which summarize load and store statements and can naturally describe local heap modifications. In this paper we assume that procedure transfer relations are supplied and we are concerned with a) verifying that transfer relations are a conservative approximation of procedure implementation b) instantiating transfer relations at call sites.

7.1 Procedure Transfer Relations

A transfer relation for a procedure proc extends the procedure signature with an initial context context(proc), and procedure effects effect(proc).

7.1.1 Initial Context

Figures 20 and 21 contain examples of initial context specification. An initial context is a description of the initial role graph $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$ where ρ_{IC} and K_{IC} are determined by a `nodes` declaration and H_{IC} is determined by a `edges` declaration. The initial role graph specifies a set of concrete heaps at procedure entry and assigns names for sets of nodes in these heaps. The next definition is similar to Definition 22.

Definition 29 *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 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 $|h_0^{-1}(n)| \leq 1$;*
4. *$h_0(\mathsf{null}_c) = \mathsf{null}$ and $h_0(\mathsf{proc}_c) = \mathsf{proc}$;*
5. *$\rho_c(o) = \rho_{\mathsf{IC}}(h_0(o))$ for every object $o \in \mathsf{nodes}(H_c)$.*

Here read(proc) is the set of initial-context nodes read by the procedure (see below). For simplicity, we assume one context per procedure; it is straightforward to generalize the treatment to multiple contexts.

A context is specified by declaring a list of nodes and a list of edges.

A list of nodes is given with `nodes` declaration. It specifies a role for every node at procedure entry. Individual nodes are denoted with lowercase identifiers, summary nodes with uppercase identifiers. By using summary nodes it is possible to indicate disjointness of entire heap regions and reachability between nodes in the heap.

There are two kinds of edges in the initial role graph: parameter edges and heap edges. A parameter edge p->pn is interpreted as $\langle \mathsf{proc}, \mathsf{p}, \mathsf{pn} \rangle \in H_{\mathsf{IC}}$. We require every parameter edge to have an individual node as a target, we call such node a *parameter node*. The role of a parameter node referenced by $\mathsf{param}_i(\mathsf{proc})$ is always $\mathsf{preR}_i(\mathsf{proc})$. Since different nodes in the initial role graph denote disjoint sets of concrete objects, parameter edges

```
p1 -> n1
p2 -> n1
```

imply that parameters `p1` and `p2` must be aliased,

```
p1 -> n1
p2 -> n2
```

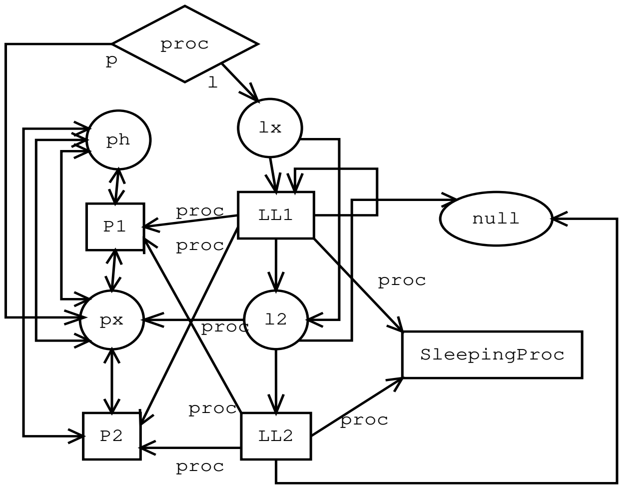
force `p1` and `p2` to be unaliased, whereas

```
p1 -> n1|n2
p2 -> n1|n2
```

allow for both possibilities. A heap edge `n -f-> m` denotes $\langle \mathtt{n}, \mathtt{f}, \mathtt{m} \rangle \in H_{\mathsf{IC}}$. The shorthand notation

```
n1 -f-> n2
   -g-> n3
```

denotes two heap edges $\langle \mathtt{n1}, \mathtt{f}, \mathtt{n2} \rangle, \langle \mathtt{n1}, \mathtt{g}, \mathtt{n3} \rangle \in H_{\mathsf{IC}}$. An expression `n1 -f-> n2|n3` denotes two edges `n1 -f-> n2` and `n1 -f-> n3`. We use similar shorthands for parameter edges.




```
nodes ph : RunningHeader,
      P1, px, P2 : RunningProc,
      lx : LiveHeader,
      LL1, l2, LL2 : LiveList;
edges p-> px, l-> px,
      ph -next-> P1|px
         -prev-> px|P2,
      P1 -next-> P1|px
         -prev-> ph|P1,
      px -next-> P2|ph
         -prev-> P1|ph,
      P2 -next-> P2|ph
         -prev-> P2|px,
      lx -next-> LL1|l2,
      LL1 -next-> LL1|l2
          -proc-> P1|P2|SleepingProc
      l2 -next-> LL2|null
         -proc-> px,
      LL2 -next-> LL2|null
          -proc-> P1|P2|SleepingProc
```

Figure 20: Initial Context for `kill` Procedure

Example 30 Figure 20 shows an initial context graph for the `kill` procedure from Example 17. It is a refinement of the role reference diagram of Figure 1 as it gives description of the heap specific to the entry of `kill` procedure. The initial context makes explicit the fact that there is only one header node for the list of running processes (`ph`) and one header node for the list of all active processes (`lx`). More importantly, it shows that traversing the list of active processes reaches a node `l2` whose `proc` field references the parameter node `px`. This is sufficient for the analysis to conclude that there will be no null pointer dereferences in the `while` loop of `kill` procedure since `l2` is reached before null.

We assume that the initial context always contains the role reference diagram RRD (Definition 8). Nodes from RRD are called *anonymous nodes* and are referred to via role name. This further reduces the size of initial context specifications by leveraging global role definitions. In Figure 20 there is no need to specify edges originating from `SleepingProc` or even mention the node `SleepingTree`, since role definitions alone contain enough information on this part of the heap to enable the analysis of the procedure. Note, however, that all edges between anonymous nodes and named nodes must be explicitly specified.

7.1.2 Procedure Effects

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

A *read effect* specifies a set read(proc) of initial graph nodes accessed by the procedure. It is used to ensure that the accessibility condition in Section 5.4.3 is satisfied. If the set of nodes denoted by read(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.

Write effects are used to modify caller's role graph to conservatively model the procedure call. A write effect $e_1.f = e_2$ approximates Store operations within a procedure. The expression e_1 denotes objects being written to, f denotes the field written, and e_2 denotes the set of objects which could be assigned to the field. Write effects are *may* effects by default, which means that the procedure is free not to perform them. It is possible to specify that a write effect *must* be performed by prefixing it with a "`!`" sign.

Example 31 In Figure 21, the `insert` procedure inserts an isolated cell into the end of an acyclic singly linked list. As a result, the role of the cell changes to `LN`. The initial context declares parameter nodes `ln` and `xn` (whose initial roles are deduced from roles of parameters), and mentions anonymous `LN` node from a default copy of the role reference diagram RRD. The code of the procedure is summarized with two write effects. The first write effect indicates that the procedure may perform zero or more Store operations to field `next` of nodes mapped to `ln` or `LN` in context(proc). The second write effect indicates that the execution of the procedure must perform a Store to the field `next` of `xn` node where the reference stored is either a node mapped onto anonymous `LN` node or `null`.

```
procedure insert(l : L,
                 x : IsolatedN ->> LN)
nodes ln, xn;
edges l-> ln, x-> xn,
      ln -next-> LN|null;
effects ln|LN . next = xn,
        ! xn.next = LN|null;
local c, p;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  p.next = x;
  x.next = c;
  setRole(x:LN);
}
```

Figure 21: Insert Procedure for Acyclic List

```
procedure insertSome(l : L)
nodes ln;
edges l-> ln,
      ln -next-> LN|null;
effects ln|LN . next = NEW,
        NEW.next = LN|null;
aux c, p, x;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  x = new;
  p.next = x;
  x.next = c;
  setRole(x:LN);
}
```

Figure 22: Insert Procedure with Object Allocation

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

Example 32 Procedure `insertSome` in Figure 22 is similar to procedure `insert` in Figure 21, except that the node inserted is created inside the procedure. It is therefore referred to in effects via generic summary node NEW.

We represent all may write effects as a set mayWr(proc) of triples $\langle n_j, f, n'_j \rangle$ where $n, n'_j \in \mathsf{nodes}_0(H_{\mathsf{IC}})$ 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 $1 \leq j \leq \mathsf{mustWrNo}(\mathsf{proc})$.

To simplify the interpretation of the declared procedure effects in terms of concrete reads and writes, we require the union $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$ to 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 7.3.2).

7.1.3 Semantics of Procedure Effects

We now give precise meaning to procedure effects. Our definition is slightly complicated by the desire to capture the set of nodes that are actually read in an execution while still allowing a certain amount of observational equivalence for write effects.

The effects of procedure proc define a subset of permissible program traces in the following way. Consider a concrete heap H_c with role assignment ρ_c such that $\langle H_c, \rho_c \rangle \, \alpha_0 \langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$ with graph homomorphism h_0 from Definition 29. Consider a trace T starting from a state with heap H_c and role assignment ρ_c. Extract the subsequence of all loads and stores in trace T. Replace Load `x=y.f` by concrete read `read` o_x where o_x is the concrete object referenced by `x` at the point of Load, and replace Store `x.f=y` by a concrete write $o_x.f = o_y$ where o_x is the object referenced by `x` and o_y object referenced by `y` at the point of Store. Let $p_1, \ldots, p_k$ be the sequence of all concrete read statements and $q_1, \ldots, q_k$ the sequence of all concrete write statements. We say that trace T starting at H_c conforms to the effects iff for all choices of h_0 the following conditions hold:

1. $h_0(o) \in \mathsf{read}(\mathsf{proc})$ for every p_i of the form `read` o
2. there exists a subsequence $q_{i_1}, \ldots, q_{i_t}$ of $q_1, \ldots, q_k$ such that
   (a) executing $q_{i_1}, \ldots, q_{i_t}$ on H_c yields the same result as executing the entire sequence $q_1, \ldots, q_k$
   (b) the sequence $q_{i_1}, \ldots, q_{i_t}$ implements write effects of procedure proc

A typical way to obtain a sequence $q_{i_1}, \ldots, q_{i_t}$ from the sequence $q_1, \ldots, q_k$ is to consider only the last write for each pair $\langle o_i, f \rangle$ of object and field.

We say that a sequence $q_{i_1}, \ldots, q_{i_t}$ implements write effects $\mathsf{mayWr}(\mathsf{proc})$ and $\mathsf{mustWr}_i(\mathsf{proc})$ for $1 \leq i \leq i_0$, $i_0 = \mathsf{mustWrNo}$ if and only if there exists an injection $s : \{1, \ldots, i_0\} \to \{i_1, \ldots, i_t\}$ such that

1. $\langle h'(o), f, h'(o') \rangle \in \mathsf{mustWr}_i(\mathsf{proc})$ for every concrete write $q_{s(i)}$ of the form $o.f = o'$, and
2. $\langle h'(o), f, h'(o') \rangle \in \mathsf{mayWr}(\mathsf{proc})$ for all concrete writes q_i of the form $o.f = o'$ for $i \in \{i_1, \ldots, i_t\} \setminus \{s(1), \ldots, s(i_0)\}$.

Here $h'(n) = h_0(n)$ for $n \in \mathsf{nodes}(H_c)$ where H_c is the initial concrete heap and $h'(n) = \mathsf{NEW}$ otherwise.

It is possible (although not very common) for a single concrete heap H_c to have multiple homomorphisms h_0 to the initial context H_{IC}. Note that in this case we require the trace T to conform to effects for *all* possible valid choices of h_0. This places the burden of multiple choices of h_0 on procedure transfer relation verification (Section 7.2) but in turn allows the context matching algorithm in Section 7.3.1 to select an arbitrary homomorphism between a caller's role graph and an initial context.

7.2 Verifying Procedure Transfer Relations

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 procedure effects, we extend the analysis representation from Section 6.1. A non-error role graph is now a tuple $\langle H, \rho, K, \tau, E \rangle$ where:

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

The initial context transformation τ tracks how objects have moved since the beginning of procedure activation and is essential for verifying procedure effects which refer to initial context 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 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.

$$
[\![\texttt{entry}\bullet]\!] = \Big\{ \langle H, \rho, K, \tau, E \rangle \;\Big|\;
\begin{array}{l}
P : \{\mathsf{proc}\} \times \{\mathsf{param}_i(\mathsf{proc})\}_i \to N, P \subseteq H_{\mathsf{IC}} \\
H_0 = (H_{\mathsf{IC}} \setminus \{\mathsf{proc}\} \times \mathsf{param}(\mathsf{proc}) \times N) \cup P \\
n_i = P(\mathsf{proc}, \mathsf{param}_i(\mathsf{proc})) \\
H_1 \subseteq H_0 \\
H_1 \setminus H_0 \subseteq \{\langle n', f, n'' \rangle \mid \{n_1, n_2\} \cap \{n_i\}_i \neq \emptyset\} \\
\forall j : \mathsf{localCheck}(n_j, \langle H, \rho, K \rangle, \mathsf{nodes}(H_1)) \\
H_1 \overset{n_1}{\parallel} H_2 \overset{n_2}{\parallel} \cdots \overset{n_p}{\parallel} H \\
\rho = \rho_{\mathsf{IC}} \\
K = K_{\mathsf{IC}} \\
\tau = \rho_{\mathsf{IC}} \\
E = \emptyset \Big\}
\end{array}
$$

Figure 23: The Set of Role Graphs at Procedure Entry

7.2.1 Role Graphs at Procedure Entry

Our role analysis creates the set of role graphs at procedure entry point from the initial context context(proc). This is simple because role graphs and the initial context have similar abstraction relations (Sections 6.1 and 7.1). The difference is that parameters in role graphs point to exactly one node, and parameter nodes are onstage nodes in role graphs which means that all their edges are "must" edges.

Figure 23 shows the construction of the initial set of role graphs. First the graph H_0 is created such that every parameter $\mathsf{param}_i(\mathsf{proc})$ references exactly one parameter node n_i. Next graph H_1 is created by using localCheck to ensure that parameter nodes have the appropriate number of edges. Finally, the instantiation is performed on parameter nodes to ensure acyclicity constraints if the initial context does not make them explicit already.

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 24: Verifying Load, Store, and New Statements

7.2.2 Verifying Basic Statements

To ensure that a procedure conforms to its transfer relation the analysis uses the initial context transformation τ to assign every Load and Store statement to a declared effect. Figure 24 shows new symbolic execution of Load, Store and New statements.

The symbolic execution of Load statement `x=y.f` makes sure that the node being loaded 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 $\tau(n_y)$ in the initial role graph context 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.

The symbolic execution of the New statement updates the initial context transformation τ assigning $\tau(n_n) = \mathsf{NEW}$ for the new node n_n.

The τ transformation is similarly updated during other abstract heap operations. Instantiation of node n' into node n_0 assigns $\tau(n_0) = \tau(n')$, split copies values of τ into the new set of isomorphic nodes, and normalization does not merge nodes n_1 and n_2 if $\tau(n_1) \neq \tau(n_2)$.

7.2.3 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 node check on all onstage nodes using predicate $\mathsf{nodeCheck}(n, \langle H, \rho, K\rangle, \mathsf{nodes}(H))$ for all $n \in \mathsf{onstage}(H)$.

At the end of the procedure, the analysis also verifies that every performed effect in $E = \{e_1, \ldots, e_k\}$ can be attributed to exactly one declared must effect. This means that $k = \mathsf{mustWrNo}(\mathsf{proc})$ and there exists a permutation s of set $\{1, \ldots, k\}$ such that $e_{s(i)} \in \mathsf{mustWr}_i(\mathsf{proc})$ for all i.

$$
\begin{array}{l}
[\![\mathsf{proc}'(x_1, \ldots, x_p)]\!](\mathcal{G}) = \\
\quad \text{if } \exists G \in \mathcal{G} : \neg\mathsf{paramCheck}(G) \text{ then } \{\bot_G\} \\
\quad\quad \text{else try } \mathcal{G}_1 = \mathsf{matchContext}(\mathcal{G}) \\
\quad\quad\quad \text{if failed then } \{\bot_G\} \\
\quad\quad\quad\quad \text{else } \{G'' \mid \langle G, \mu\rangle \in \mathcal{G}_1 \\
\quad\quad\quad\quad\quad \langle \mathsf{addNEW}(G), \mu\rangle \xrightarrow{\mathsf{FX}} \langle G', \mu\rangle \xrightarrow{\mathsf{RR}} G''\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{paramCheck}(\langle H, \rho, K, \tau, E\rangle) \text{ iff} \\
\quad \forall n_i : \mathsf{nodeCheck}(n_i, G, \mathsf{offstage}(H) \cup \{n_i\}_i) \\
\quad n_i \text{ are such that } \langle \mathsf{proc}, x_i, n_i\rangle \in H
\end{array}
$$

$$
\begin{array}{l}
\mathsf{addNEW}(\langle H, \rho, K, \tau, E\rangle) = \\
\quad \langle H \cup \{n_0\} \times F \times \{\mathsf{null}\}, \\
\quad \rho[n_0 \mapsto \mathsf{unknown}], \\
\quad K[n_0 \mapsto s], \\
\quad \tau[n_0 \mapsto \mathsf{NEW}], \\
\quad E\rangle \\
\text{where } n_0 \text{ is fresh in } H
\end{array}
$$

Figure 25: Procedure Call

7.3 Analyzing Call Sites

The set of role graphs at the procedure call site is updated based on the procedure transfer relation as follows. Consider procedure proc containing call site $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ with procedure call $\mathsf{proc}'(x_1, \ldots, x_p)$. Let $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle = \mathsf{context}(\mathsf{proc}')$ be the initial context of the callee.

Figure 25 shows the transfer function for procedure call sites. It has the following phases:

1. **Parameter Check** ensures that roles of parameters conform to the roles expected by the callee proc'.
2. **Context Matching** ($\mathsf{matchContext}$) ensures that the caller's role graphs represent a subset of concrete heaps represented by $\mathsf{context}(\mathsf{proc}')$. This is done by deriving a mapping μ from the caller's role graph to $\mathsf{nodes}(H_{\mathsf{IC}})$.

3. **Effect Instantiation** ($\xrightarrow{\textsf{FX}}$) uses effects $\textsf{mayWr}(\textsf{proc}')$ and $\textsf{mustWr}_i(\textsf{proc}')$ in order to approximate all structural changes to the role graph that $\textsf{proc}'$ may perform.

4. **Role Reconstruction** ($\xrightarrow{\textsf{RR}}$) uses final roles for parameter nodes and global role declarations $\textsf{postR}_i(\textsf{proc}')$ to reconstruct roles of all nodes in the part of the role graph representing modified region of the heap.

The parameter check requires $\textsf{nodeCheck}(n_i, G, \textsf{offstage}(H) \cup \{n_i\}_i)$ for the parameter nodes n_i. The other three phases are explained in more detail below.

7.3.1 Context Matching

Figure 26 shows our context matching function. The $\textsf{matchContext}$ 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 $H_{\textsf{IC}}$. The homomorphism μ guarantees that $\alpha^{-1}(G) \subseteq \alpha_0^{-1}(\textsf{context}(\textsf{proc}'))$ since the homomorphism h_0 from Definition 29 can be constructed from homomorphism h in Definition 22 by putting $h_0 = \mu \circ h$. This implies that it is legal to call $\textsf{proc}'$ with any concrete graph represented by G.

The algorithm in Figure 26 starts with empty maps $\mu = \textsf{nodes}(G) \times \{\bot\}$ and extends μ until it is defined on all $\textsf{nodes}(G)$ 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 context that n_0 can be mapped to. The candidates are chosen to make sure that μ remains a homomorphism. The accessibility requirement—that a procedure may see no nodes with incorrect role—is enforced by making sure that nodes in $\textsf{inaccessible}$ are never mapped into nodes in $\textsf{read}$ for the callee. As long as this requirement holds, nodes in $\textsf{inaccessible}$ 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 context 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 by performing instantiation of a summary node of 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 $\textsf{match}$ in Figure 26).

The algorithm is nondeterministic in the order in which nodes to be matched are selected. One possible ordering of nodes is 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 procedure call lexicographically increases the sorted list of numbers $|\mu[\textsf{nodes}(H)]|$ for $\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle \in \Gamma$.

7.3.2 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 procedure effects in each of the role graphs of the caller. Figure 30 gives rules for effect instantiation. The analysis first verifies that the region read by the callee is included in the region read by the caller. Then it uses map μ to find the inverse image S of the performed effects. The effects in S are grouped by the source n and field f. Each field $n.f$ is applied in sequence. There are three cases when applying an effect to $n.f$:

1. There is only one node target of the write in $\textsf{nodes}(H)$ and the effect is a must write effect. In this case we do a strong update.
2. The condition in 1) is not satisfied, and the node n is offstage. In this case we conservatively add all relevant edges from S to H.
3. The condition in 1) is not satisfied, but the node n is onstage i.e. it is a parameter node[3]. In this case there is no unique target for $n.f$, and we cannot add multiple edges either as this would violate the invariant for onstage nodes. We therefore do case analysis choosing which effect was performed last. If there are no must effects that affect n, then we also consider the case where the original graph is unchanged.

7.3.3 Role Reconstruction

Procedure effects approximate structural changes to the heap, but do not provide information about role changes for non-parameter nodes. We use the role reconstruction algorithm $\xrightarrow{\textsf{RR}}$ in Figure 27 to conservatively infer possible roles of nodes after the procedure call based on role changes for parameters and global role definitions.

Role reconstruction first finds the set N_0 of all nodes that might be accessed by the callee since these nodes might have their roles changed. Then it 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 procedure executes. The edges between nodes in the new graph are derived by simultaneously satisfying 1) structural constraints between nodes of the original graph; and 2) global role constraints from the role reference diagram. The nodes $\rho(n, r)$ not connected to the parameter nodes are garbage collected in the role graph. In practice, we generate nodes $\rho(n, r)$ and edges on demand starting from parameters making sure that they are reachable and satisfy both kinds of constraints.

8 Extensions

This section presents two extensions of the basic role system. The first extension allows statically unbounded number of aliases for objects. The second extension allows the analysis to verify more complex role changes. Additional ways of extending roles are given in [31].

8.1 Multislots

A multislot $\langle r', f\rangle \in \textsf{multislots}(r)$ in the definition of role r allows any number of aliases $\langle o', f, o\rangle \in H_c$ for $\rho_c(o') = r'$ and $\rho_c(o) = r$. We require multislots $\textsf{multislots}(r)$ to be disjoint from all $\textsf{slot}_i(r)$. To handle multislots in role analysis we relax the condition 5) in Definition 22 of the abstraction

[3]Non-parameter onstage nodes are never affected by effects, as guaranteed by the matching algorithm.

$$
\begin{array}{l}
\quad \mathsf{matchContext}(\mathcal{G}) = \mathsf{match}(\{\langle G, \mathsf{nodes}(G) \times \{\bot\}\rangle \mid G \in \mathcal{G}\}) \\[1em]
\mathsf{match} : \mathcal{P}(\mathsf{RoleGraphs} \times (N \cup \{\bot\})^N) \rightharpoonup \mathcal{P}(\mathsf{RoleGraphs} \times N^N) \\[1em]
\mathsf{match}(\Gamma) = \\
\quad \Gamma_0 := \{\langle G, \mu\rangle \in \Gamma \mid \mu^{-1}(\bot) \neq \emptyset\}; \\
\quad \mathsf{if}\ \Gamma_0 = \emptyset\ \mathsf{then}\ \ \mathsf{return}\ \Gamma; \\
\quad \langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle := \ \mathsf{choose}\ \Gamma_0; \\
\quad \Gamma' = \Gamma \setminus \langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle; \\
\quad \mathsf{paramnodes} := \{n \mid \exists i : \langle \mathsf{proc}, x_i, n\rangle \in H\}; \\
\quad \mathsf{inaccessible} := \mathsf{onstage}(H) \setminus \mathsf{paramnodes}; \\
\quad n_0 := \ \mathsf{choose}\ \mu^{-1}(\bot); \\
\quad \mathsf{candidates} := \{n' \in \mathsf{nodes}(H_{\mathsf{IC}}) \mid \\
\qquad\qquad (n_0 \notin \mathsf{inaccessible}\ \text{and}\ \rho_{\mathsf{IC}}(n') = \rho(n_0))\ \text{or} \\
\qquad\qquad (n_0 \in \mathsf{inaccessible}\ \text{and}\ n' \notin \mathsf{read}(\mathsf{proc}'))\} \\
\qquad \displaystyle\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\} \\
\qquad \displaystyle\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\}; \\
\quad \mathsf{if\ candidates} = \emptyset\ \mathsf{then}\ \ \mathsf{fail}\ ; \\
\quad \mathsf{if\ candidates} = \{n_0'\}, K(n_0) = s, K_{\mathsf{IC}}(n_0') = i, \mu^{-1}(n_0') = \emptyset \\
\qquad \mathsf{then\ match}(\Gamma' \cup \{\langle G', \mu[n_1 \mapsto n_0']\rangle \mid \langle H, \rho, K, \tau, E\rangle \overset{n_1}{\underset{n_0}{\Uparrow}} G'\}) \\
\quad\ \mathsf{else}\ n_0' := \ \mathsf{choose}\ \{n' \in \mathsf{candidates} \mid K(n') = s\ \text{or} \\
\qquad\qquad\qquad (K(n_0) = i, \mu^{-1}(n') = \emptyset)\} \\
\qquad\quad \mathsf{match}(\Gamma' \cup \langle\langle H, \rho, K, \tau, E\rangle, \mu[n_0 \mapsto n_0']\rangle);
\end{array}
$$

Figure 26: The Context Matching Algorithm

relation by allowing h to map more than one concrete edge $\langle o', f, o\rangle$ onto abstract edge $\langle n', f, n\rangle \in H$ terminating at an onstage node n provided that $\langle \rho(n'), f\rangle \in \mathsf{multislots}(\rho(n))$. The nodeCheck and expansion relation $\preceq$ are then extended appropriately. Note that a role graph does not represent the exact number of references that fill each multislot. The analysis therefore does not attempt to recognize actions that remove the last reference from the multislot. Once an object plays a role with a multislot, all subsequent roles that it plays must also have the multislot.

8.2 Cascading Role Changes

In some cases it is desirable to change 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 such *cascading role change* of a set of nodes. The need for cascading role changes arises when roles encode reachability properties.

Example 33 The code fragment in Figure 28 manipulates object `m` of role `Data`. The role `Data` has fields `buffer` and `work`, each being a root for a singly linked acyclic list. Elements of the first list have `BufferNode` role and elements of the second list have `WorkNode` role. At some point procedure swaps the contents of the fields `buffer` and `work`, which requires all nodes in both lists to change the roles. These role changes are triggered by the `setRoleCascade` statement.

```
role BufferNode {
  fields next : BufferNode | null;
  slots  BufferNode.next | Data.buffer;
  acyclic next;
}
role WorkNode {
  fields next : WorkNode | null;
         WorkNode.next | Data.work;
  acyclic next;
}
role Data {
  fields buffer : BufferNode | null,
         work   : WorkNode | null;
}

{
   ...
   roleCheck(m : Data);
   x = m.buffer;
   y = m.work;
   m.buffer = y;
   m.work = x;
   setRoleCascade(x:WorkNode, y:BufferNode);
   ...
}
```

Figure 28: Example of a Cascading Role Change

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{RR}} \langle H',\rho',K',\tau',E'\rangle$$

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

Figure 27: Call Site Role Reconstruction

The statement indicates new roles for onstage nodes, and the analysis cascades role changes to offstage nodes.

Given a role graph $\langle H,\rho,K,E\rangle$ cascading role change finds a new valid role assignment ρ' where the onstage nodes have desired roles and the roles of offstage nodes are adjusted appropriately. Figure 29 shows abstract execution of the `setRoleCascade` statement. Here $\mathsf{neighbors}(n,H)$ denotes nodes in H adjacent to n. The condition $\mathsf{cascadingOk}(n,H,\rho,K,\rho')$ makes sure it is legal to change the role of 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 for `setRole` statement in Section 6.2.3. Let $r = rho(n)$ and $r' = \rho'(n)$. Then $\mathsf{cascadingOk}(n,H,\rho,K,\rho')$ requires the following conditions:

1. $\langle n,f,n_1\rangle \in H$ implies $\rho'(n_1) \in \mathsf{field}_f(r')$
2. $\mathsf{slotno}(r') = \mathsf{slotno}(r) = k$, and for every list $\langle n_1,f_1,n\rangle,\ldots,\langle n_k,f_k,n\rangle \in H$ if there is a permutation $p : \{1,\ldots,k\} \to \{1,\ldots,k\}$ such that $\langle\rho(n_i),f_i\rangle \in \mathsf{slot}_{p_i}(r)$, then there is a permutation $p' : \{1,\ldots,k\} \to \{1,\ldots,k\}$ such that $\langle\rho(n_i),f_i\rangle \in \mathsf{slot}_{p_i}(r')$.
3. identity relations were already satisfied or can be explicitly checked: $\langle f,g\rangle \in \mathsf{identities}(\rho'(n))$ implies
   (a) $\langle f,g\rangle \in \mathsf{identities}(\rho(n))$ or
   (b) for all $\langle n,f,n'\rangle \in H$: $K(n') = i$, and if $\langle n',g,n''\rangle \in H$ then $n'' = n$
4. either $\mathsf{acyclic}(\rho'(n)) \subseteq \mathsf{acyclic}(\rho(n))$ or $\mathsf{acycCheck}(n,\langle H,\rho',K\rangle,\mathsf{offstage}(H))$.

In practice there may be zero or more solutions that satisfy constraints for a given cascading role change. Selecting any solution that satisfies the constraints is sound with respect to the original semantics. A useful heuristic for searching the solution space is to first explore branches with as few roles changed as possible. If no solutions are found, an error is reported.

9 Related Work

Typestate, as a type system extension for statically verifying dynamically changing properties, was proposed in [44, 43]. Aliasing causes problems for typestate-based systems because the declared typestates of all aliases must change whenever the state of the referred object changes. Faced with the complexity of aliasing, [44] resorted to a more controlled language model which avoids aliasing. More recently proposed typestate approaches use linear types for heap references to support state changes of dynamic allocated objects without addressing aliasing issues [10].

Motivated by the need to enforce safety properties in low-level software systems, [42, 46, 9] use extensions of linear types to describe aliasing of objects and rely on language design to avoid non-local type inference. These systems take a *construction based approach* that specifies data structures as unfoldings of basic elaboration steps [46]. Similarly to shape types [15, 14] and graph types [29, 34], this allows tree-like data structures to be expressed more precisely than using our roles, but cannot approximate data structures such as sparse matrices. More importantly, this approach makes it difficult to express nodes that are members of multiple data structures. Handling multiple data structures is the essential ingredient of our approach because the role of an object depends on data structures in which it participates.

Like shape analysis techniques [5, 17, 39, 40] we have therefore adopted the *constraint based approach* which characterizes data structures in terms of the constraints that they satisfy. The constraint based approach allows us to handle a wider range of data structure while giving up some precision. Like [47, 48] we perform non-local inference of program properties, but while [47, 48] focus on linear integer constraints and handle recursive data structures conservatively, we do not handle integer arithmetic but have a more precise representation of the heap. At a higher level, these approaches all focus on detailed properties of individual data structures. We view our research as focusing more on global aspects such as the participation of objects in multiple data structures.

The path matrix approaches [18, 17] have been used to implement efficient interprocedural analyses that infer one level of referencing relationships, but are not sufficiently pre-

$\langle H, \rho, K, \tau, E\rangle \overset{\mathsf{s}}{\rightsquigarrow} \langle H, \rho', K, \tau, E\rangle$ $s = \texttt{setRoleCascade}(x_1 : r_1, \ldots, x_n : r_n)$	$n_i : \langle \mathsf{proc}, x_i, n_i\rangle \in H$ $\rho'(n_i) = r_i$ $\rho'(n) = \rho(n),\ n \in \mathsf{onstage}(H) \setminus \{n_i\}_i$ $N_0 = \{n \in \mathsf{offstage}(H) \mid \exists n' \in \mathsf{neighbors}(n, H) : \rho(n') \neq \rho'(n')\}$ $\forall n \in N_0 : \mathsf{cascadingOk}(n, H, \rho, K, \rho')$

Figure 29: Abstract Execution for `setRoleCascade`

cise to track must aliases of heap objects for programs with destructive updates of more complex data structures.

The use of the instantiation relation in role analysis is analogous to the materialization operation of [39, 40]. Role analysis can also track reachability properties, but we use an abstraction relation based on graph homomorphism rather than 3-valued logic. Our split operation achieves a similar goal to the focus operation of [40]. However, the generic focus algorithm of [32] cannot handle the reachability predicate which is needed for our split operation. This is because it conservatively refuses to focus on edges between two summary nodes to avoid generating an infinite number of structures. Rather than requiring definite values for reachability predicate, our role analysis splits by reachability properties in the abstract role graph, which illustrates the flexibility of the homomorphism-based abstraction relation. Another difference with [40] is that our role analysis does not require the developer to supply the predicate update formulae for instrumentation predicates.

A precise interprocedural analysis [38] extends shape analysis techniques to treat activation records as dynamically allocated structures. The approach also effectively synthesizes an application-specific set of contexts. 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 to verify that it conforms to its specification. Like [48] our interprocedural analysis can apply both may and must effects, but our contexts are general graphs with summary nodes and not trees.

Roles are similar to the ADDS and ASAP data structure description languages [25, 26, 23]. These systems use sound techniques to apply the data structure invariants for parallelization and general dependence testing but do not verify that the data structure invariants are preserved by destructive updates of data structures [24].

The object-oriented community has long been aware of benefits that dynamically changing classes give in large systems [37]. Recognizing these benefits, researchers have proposed dynamic techniques that change the class of an object to reflect its state changes [16, 20, 4, 13]. These systems illustrate the need for a static system that can verify the correct use of objects with changing roles.

10 Conclusion

This paper proposes two key ideas: aliasing relationships should determine, in large part, the state 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 in a concrete system, and present an analysis algorithm that can verify that the program correctly respects the constraints of this role system. 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.

Note. This November 2001 version of the technical report corrects some errors in the formal description of the analysis algorithm of the original technical report from July 2001.

REFERENCES

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[2] Barendsen and J. E. W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 1993.

[3] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, number 6 in 26, pages 278–292, 1991.

[4] Craig Chambers. Predicate classes. In *Proc. 7th ECOOP*, pages 268–296, 1993.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.

[6] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In *Proc. 26th ACM POPL*, pages 133–146, 1999.

[7] Ben-Chung Cheng and Wen mei W. Hwu. Modular interprocedural pointer analysis using access paths. In *Proc. ACM PLDI*, 2000.

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

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{FX}} \langle\bot_G,\mu\rangle \text{ where } \quad \tau[\mu^{-1}[\mathsf{read}(\mathsf{proc}')]] \not\subseteq \mathsf{read}(\mathsf{proc})$$

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{FX}} G_t \text{ where } \quad \begin{array}{l}\tau[\mu^{-1}[\mathsf{read}(\mathsf{proc}')]] \subseteq \mathsf{read}(\mathsf{proc}) \\ \langle H,\rho,K,\tau,E\rangle \overset{n_1,f_1}{\vdash} G_1 \vdash \cdots \overset{n_t,f_t}{\vdash} G_t\end{array}$$

$$S = \{\langle n,f,n'\rangle \in H \mid \langle\mu(n),f,\mu(n')\rangle \in \mathsf{mayWr}(\mathsf{proc}') \cup \cup_i \mathsf{mustWr}_i(\mathsf{proc}')\}$$

$$\{\langle n_1,f_1\rangle,\ldots,\langle n_t,f_t\rangle\} = \{\langle n,f\rangle \mid \langle n,f,n'\rangle \in S\}$$

Single Write Effect Instantiation:

$$\langle H_1,\rho_1,K_1,\tau_1,E_1\rangle \overset{n,f}{\vdash} G'$$

iff

case	condition	result
deterministic effect	$\{n_1 \mid \langle n,f,n_1\rangle \in S\} = \{n_0\}$ and $\exists i : \langle\mu(n),f,\mu(n_0)\rangle \in \mathsf{mustWr}_i(\mathsf{proc}')$	$G' = \langle H_2,\rho_1,K_1,\tau_1,E_2\rangle$ $H_2 = H_1 \setminus \{\langle n,f,n_1\rangle \mid \langle n,f,n_1\rangle \in H_1\}$ $\cup\{\langle n,f,n_0\rangle\}$ $E_2 = \mathsf{updateWr}(E_1, \langle\tau(n),f,\tau(n_0)\rangle)$
nondeterministic effect for non-parameters	$\lvert\{n_1 \mid \langle n,f,n_1\rangle \in S\}\rvert > 1$ or $\exists n_1 : \langle\mu(n),f,\mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \in \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \langle H_2,\rho_1,K_1,\tau_1,E_2\rangle$ $H_2 = \mathsf{orem}(H_1)\cup$ $\{\langle n,f,n_1\rangle \mid \langle n,f,n_1\rangle \in S\}$
	$\lvert\{n_1 \mid \langle n,f,n_1\rangle \in S\}\rvert > 1$ or $\exists n_1 : \langle\mu(n),f,\mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \in \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \not\subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \bot_G$
nondeterministic effect for parameters	$\lvert\{n_1 \mid \langle n,f,n_1\rangle \in S\}\rvert > 1$ or $\exists n_1 : \langle\mu(n),f,\mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \notin \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \langle H_2,\rho_1,K_1,\tau_1,E_2\rangle$ $H_0 = H_1 \setminus \{\langle n,f,n_1\rangle \mid \langle n,f,n_1\rangle \in H_1\}$ $H_2 = H_1$ or $H_2 = H_0 \cup \{\langle n,f,n_1\rangle\}$ $\langle n,f,n_1\rangle \in S$
	$\neg(\{n_1 \mid \langle n,f,n_1\rangle \in S\} = \{n_1\}$ and $\exists i : \langle\mu(n),f,\mu(n_0)\rangle \in \mathsf{mustWr}_i(\mathsf{proc}'))$ $n \notin \mathsf{offstage}(H)$ $\{\langle\tau(n),f,\tau(n_1)\rangle \mid \langle n,f,n_1\rangle \in S\} \not\subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \bot_G$

$$\mathsf{orem}(H_1) = \begin{cases} H_1 \setminus \{\langle n,f,n'\rangle \mid \langle n,f,n'\rangle \in H_1\}, & \text{if } \exists i\, \exists n' : \langle\mu(n),f,\mu(n')\rangle \in \mathsf{mustWr}_i(\mathsf{proc}') \\ H_1, & \text{otherwise}\end{cases}$$

Figure 30: Effect Instantiation

[9] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM POPL*, 1999.

[10] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

[11] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, 1994.

[12] Amer Diwan, Kathryn McKinley, and J. Elliot B. Moss. Type-based alias analysis. In *Proc. ACM PLDI*, 1998.

[13] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th ECOOP*, LNCS 2072, pages 130–149. Springer, 2001.

[14] Pascal Fradet and Daniel Le Metayer. Structured gamma. Technical Report 989, IRISA, 1996.

[15] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[17] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

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

[19] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proc. 25th ACM POPL*, 1998.

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

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

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

[23] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM PLDI*, 1994.

[24] 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.

[25] 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.

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

[27] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.

[28] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM POPL*, 1991.

[29] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[30] Naoki Kobayashi. Quasi-linear types. In *Proc. 26th ACM POPL*, 1999.

[31] Viktor Kuncak. Designing an algorithm for role analysis. Master's thesis, MIT Laboratory for Computer Science, 2001.

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

[33] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation, 121(2)*, 1995.

[34] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[35] John Plevyak, Vijay Karamcheti, and Andrew A. Chien. Analysis of dynamic structures for efficient parallel execution. In *Workshop on Languages and Compilers for Parallel Architectures*, 1993.

[36] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990.

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

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

[39] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. 23rd ACM POPL*, 1996.

[40] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th ACM POPL*, 1999.

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

[42] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. 9th ESOP*, Berlin, Germany, March 2000.

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

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

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

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

[47] Zhichen Xu, Barton Miller, and Thomas Reps. Safety checking of machine code. In *Proc. ACM PLDI*, 2000.

[48] Zhichen Xu, Thomas Reps, and Barton Miller. Typestate checking of machine code. In *Proc. 10th ESOP*, 2001.