A Specification-Based Approach to Reasoning about Pointers

Gregory Kulczycki
Virginia Tech

Murali Sitaraman
Clemson University

Bruce W. Weide
The Ohio State University

Atanas Rountev
The Ohio State University
Motivation

Traditional static analysis techniques for pointers are limited in what they can prove about pointer programs.
Example: Shape Analysis

```c
typedef struct list {
    struct list *n;
    int data;
} List;

List *splice(List *x, List *y) {
    List *t = NULL;
    List *z = y;
    while (x != NULL) {
        t = x;
        x = t->n;
        t->n = y->n;
        y->n = t;
        y = y->n->n;
    }
    return z;
}
```
Example: Shape Analysis

Example state at the beginning of splice

Example state at the end of splice
Example: Shape Analysis

“Statically verify that, if the input lists x and y are disjoint and acyclic, then the list returned by splice is acyclic.”

Region-based shape analysis with tracked locations
Hackett and Rugina, POPL 2005
Example: Shape Analysis

“Statically verify that, if the input lists \( x \) and \( y \) are disjoint and acyclic, then the list returned by \( \text{splice} \) is acyclic.”

Region-based shape analysis with tracked locations
Hackett and Rugina, POPL 2005
Example: Shape Analysis

Shape analysis helps with some properties:

- Is a memory location referenced by more than one other location?
- Is a location accessed through a dangling references?
- Are memory leaks present?
Example: Shape Analysis

Shape analysis helps with some properties:

• Is a memory location referenced by more than one other location?
• Is a location accessed through a dangling references?
• Are memory leaks present?

But it does not help with other properties:

• Does the splice operation do what it is supposed to do (i.e., does the operation interleave the elements of the incoming lists)?
Specification-Based Approach

- Provide a generic pointer component
- Include its full formal specification
- Give it a special implementation
- Special syntax is optional
Concept Location_Linking_Template (type Info);

Defines Location: Set;
Defines Void: Location;
Var Target: Location → Location;
Var Contents: Location → Info;
Var Is_Taken: Location → B;

Initialization ensures ∀q: Location, ¬Is_Taken(q);

Type Position is modeled by Location;
  exemplar p;
  Initialization ensures p = Void;
...

Mathematical Model

**Concept** Location_Linked_Template (type Info);

**Defines** Location: Set;
**Defines** Void: Location;
**Var** Target: Location → Location;
**Var** Contents: Location → Info;
**Var** Is_Taken: Location → B;

**Initialization ensures** ∀q: Location, ¬Is_Taken(q);

**Type** Position is modeled by Location;
  **exemplar** p;
  **Initialization ensures** p = Void;

...
Concept Location_Linking_Template (type Info);

Defines Location: Set;
Defines Void: Location;
Var Target: Location \rightarrow Location;
Var Contents: Location \rightarrow Info;
Var Is_Taken: Location \rightarrow B;

Initialization ensures \forall q: Location, \neg Is\ Taken(q);

Type Position is modeled by Location;
   exemplar p;
   Initialization ensures p = Void;

...
A System of Linked Locations

free, non-Void locations

taken locations

\( b \)

\( \alpha \)  \( \beta \)

\( \chi \)  \( \delta \)

\( \lambda \)

\( y, z \)
Concept Location_Linking_Template (type Info);
Type Position;
Operation Take_New_Location (updates p: Position);
Operation Abandon_Location (clears: p: Position);
Operation Relocate (updates p: Position;
preserves q: Position);
Operation Follow_Link (updates p: Position);
Operation Redirect_Link (preserves p: Position;
preserves q: Position);
...
end Location_Linking_Template;
Redirect Link

**Operation** Redirect_Link(*preserves* p: Position;  
*preserves* q: Position);  

*updates* Target;  
*requires* Is_Taken(p);  
*ensures* ∀r: Location,

\[
\text{Target}(r) = \begin{cases} 
q & \text{if } r = p \\
\#\text{Target}(r) & \text{otherwise} 
\end{cases};
\]
before

$Redirect\_Link(p, q)$
before

\textit{Redirect\_Link}(p, q)
before

\[ p = \text{WAS} \]
\[ q = \text{PAR} \]

\text{Redirect}\_\text{Link}(p, q)
\( \text{Redirect\_Link}(p, q) \)

\[ \text{Target} = \{ \text{WAS} \leftrightarrow \text{PAR}, \ldots \} \]
\[ \text{Contents} = \{ \text{WAS} \leftrightarrow \alpha, \]
\[ \quad \text{PAR} \leftrightarrow \beta, \]
\[ \quad \text{LIS} \leftrightarrow \delta, \ldots \} \]
\[ \text{Is\_Taken} = \{ \text{WAS} \leftrightarrow \text{true}, \]
\[ \quad \text{PAR} \leftrightarrow \text{true}, \]
\[ \quad \text{LIS} \leftrightarrow \text{true}, \ldots \} \]

\( p = \text{WAS} \)
\( q = \text{PAR} \)
Relocate$(p, q)$
Relocate \((p, q)\) after
before

Follow_Link(p)
Follow_Link(p)

after

p, q

α

β
Implementation

Operation invocations such as

- `Relocate(p, q)`
- `Follow_Link(p)`
- `Redirect_Link(p, q)`

are implemented internally by copying a memory address, not by invoking an operation.
Splice Operation

**Operation** Splice (**preserves** p: Position; **clears** q: Position );

**updates** Target;

- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p’s list is at least as long as q’s
- **postcondition:** p’s resulting list is an interleaving of q’s incoming list with the first locations of p’s incoming list.
Operation Splice (preserves p: Position; clears q: Position);
updates Target;

- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p’s list is at least as long as q’s
- **postcondition:** p’s resulting list is an interleaving of q’s incoming list with the first locations of p’s incoming list.

before
Operation Splice (preserves p: Position; clears q: Position);
updates Target;

• **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p’s list is at least as long as q’s

• **postcondition:** p’s resulting list is an interleaving of q’s incoming list with the first locations of p’s incoming list.
Definitions

Is\_Reachable\_in (hops: \textbf{N}; p, q: Location): \textbf{B}

Is\_Reachable (p, q: Location): \textbf{B}

Distance(p, q: Location): \textbf{N}
Definitions

\[ \text{Is\_Reachable\_in} \text{ (hops: } N; p, q: \text{ Location}) : B \]
\[ \text{Is\_Reachable} \text{ (p, q: Location): } B \]
\[ \text{Distance(p, q: Location): } N \]

Is\_Reachable\_in(3, p, q)
Definitions

Is_Reachable_in (hops: \textbf{N}; p, q: Location): \textbf{B}

Is_Reachable (p, q: Location): \textbf{B}

Distance(p, q: Location): \textbf{N}

Is_Reachable_in(3, p, q)
Is_Reachable(p, Void)
Definitions

Is_Reachable_in (hops: \textbf{N}; p, q: Location): \textbf{B}

Is_Reachable (p, q: Location): \textbf{B}

Distance(p, q: Location): \textbf{N}

Is_Reachable_in(3, p, q)
Is_Reachable(p, Void)
Distance(q, Void) = 1
Lightweight Specification

**Operation** Splice (**preserves** p: Position; **clears** q: Position);
**updates** Target;
**requires** ( \( \exists k_1, k_2 : \mathbb{N} \ \exists \) )
\[ \text{Is\_Reachable\_in}(k_1, p, \text{Void}) \text{ and } \text{Is\_Reachable\_in}(k_2, q, \text{Void}) \text{ and } k_2 \leq k_1 \] and
( \( \forall r : \text{Location}, \) )
\[
\text{if Is\_Reachable}(p, r) \text{ and Is\_Reachable}(q, r) \text{ then } r = \text{Void} ;
\]
**ensures** Is\_Reachable(p, Void);
**Operation** Splice *(preserves* p: Position; *clears* q: Position*);*

**updates** Target;

**Procedure**

Var r: Position;
Var s: Position;
Relocate(r, p);
While *(not* Is_At_VOID(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
end Splice;
While (not Is_At_Void(q))
  do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
  end;
While (not Is_AtVoid(q))
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
While (not Is_At_Void(q))
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
While (not Is_At_Void(q))
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
While (not Is_At.Void(q))
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
While (not Is_At_Void(q))
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
While (not Is_At_VOID(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
While (not Is_AtVoid(q))
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
Splice Procedure

**Procedure**

Var r: Position;
Var s: Position;
Relocate(r, p);
While (not Is_AtVoid(q))

**maintaining ???**
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
end Splice;

**Operation** Splice (**preserves** p: Position; **clears** q: Position);

- **updates** Target;
- **requires** ( ∃ k₁, k₂ : N ∃
  - Is_Reachable_in(k₁, p, Void) and
  - Is_Reachable_in(k₂, q, Void) and
  - k₂ ≤ k₁ ) and
  ( ∀ r : Location,
  - if Is_Reachable(p, r) and Is_Reachable(q, r)
  - then r = Void );
- **ensures** Is_Reachable(p, Void);
### Splice Procedure

**Procedure**

```plaintext
Var r: Position;
Var s: Position;
Relocate(r, p);
While (not Is_AtVoid(q))
    maintaining Is_Reachable(p, Void);
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
end Splice;
```

**Operation** `Splice` *(preserves p: Position; clears q: Position)*:

- **updates** Target;
- **requires** *(∃ k₁, k₂ : N) (Is_Reachable_in(k₁, p, Void) and Is_Reachable_in(k₂, q, Void) and k₂ ≤ k₁) and (∀ r : Location, if Is_Reachable(p, r) and Is_Reachable(q, r) then r = Void)*;
- **ensures** Is_Reachable(p, Void);
Splice Procedure

**Procedure**

Var \( r: \text{Position}; \)

Var \( s: \text{Position}; \)

Relocate\((r, p)\);

While (not \( \text{Is\_At\_Void}(q) \))

**decreasing ???**

**maintaining** \( \text{Is\_Reachable}(p, \text{Void}); \)

do

Relocate\((s, r)\);

Follow\_Link\((r)\);

Redirect\_Link\((s, q)\);

Follow\_Link\((s)\);

Follow\_Link\((q)\);

Redirect\_Link\((s, r)\);

end;

end Splice;

**Operation** Splice (preserves \( p: \text{Position}; \)

 clears \( q: \text{Position}; \))

updates Target;

requires ( \( \exists \ k_1, k_2: \mathbb{N} \ \exists \)

 \( \text{Is\_Reachable\_in}(k_1, p, \text{Void}) \ and \)

 \( \text{Is\_Reachable\_in}(k_2, q, \text{Void}) \ and \)

 \( k_2 \leq k_1 \) and

( \( \forall \ r: \text{Location}, \)

 if \( \text{Is\_Reachable}(p, r) \ and \text{Is\_Reachable}(q, r) \)

 then \( r = \text{Void} \));

ensures \( \text{Is\_Reachable}(p, \text{Void}); \)
Splice Procedure

**Procedure**

Var r: Position;
Var s: Position;
Relocate(r, p);
While (not Is_At.Void(q))
  decreasing Distance(q, Void);
  maintaining Is.Reachable(p, Void);
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
end Splice;

**Operation** Splice (preserves p: Position; clears q: Position);

updates Target;
requires ( ∃ k₁, k₂ ∈ ℕ ∃ r : Location,
  Is.Reachable_in(k₁, p, Void) and
  Is.Reachable_in(k₂, q, Void) and
  k₂ ≤ k₁ ) and
( ∀ r : Location,
  if Is.Reachable(p, r) and Is.Reachable(q, r)
  then r = Void );
ensures Is.Reachable(p, Void);
Loop Invariant Proof

Procedure

Var r: Position;
Var s: Position;
Relocate(r, p);
While (not Is_At_Void(q))
  decreasing Distance(q, Void);
  maintaining Is_Reachable(p, Void);
do
  Relocate(s, r);
  Follow_Link(r);
  Redirect_Link(s, q);
  Follow_Link(s);
  Follow_Link(q);
  Redirect_Link(s, r);
end;
end Splice;

1. Initialization – Is the invariant true at the start of the loop?
2. Maintenance – Is the invariant true from one iteration to the next?
3. Termination – Does the invariant allow you to prove what you need to?
Loop Invariant Proof

Lemma #1: Is_Reachable(q, Void);
Lemma #2: Is_Reachable(r, Void);
Lemma #3: Is_Reachable(p, r) or Is_Reachable(p, q);

While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
Operation Splice (preserves p: Position; clears q: Position);
updates Target;

- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p’s list is at least as long as q’s
- **postcondition:** p’s resulting list is an interleaving of q’s incoming list with the first locations of p’s incoming list.
String Notation

The following notations are defined in a module that specifies the properties of mathematical strings.

\( \langle x \rangle \)  
a string containing \( x \)

\( \alpha \circ \beta \)  
the concatenation of strings \( \alpha \) and \( \beta \)

\( \alpha \leq!\geq (\beta, \gamma) \)  
asserts that \( \alpha \) is a perfect shuffle of strings \( \beta \) and \( \gamma \)
Definitions

\[ \text{Is\_Info\_Str}(p, q: \text{Location}, \alpha: \text{Str(Info)}: B) \]
Is_Info_Str(p, q: Location, α: Str(Info): B)

Is_Info_Str(p, q, ⟨α, β, γ⟩)
**Heavyweight Specification**

**Operation** Splice (**preserves** p: Position; **clears** q: Position);

**updates** Target;

**requires** **Same as in lightweight specification**

**ensures** ( ∀ t: Location, if not Is_Reachable(#p, t) and not Is_Reachable(#q, t) then Target(t) = #Target(t) ) and

( ∀ po, old_po, old_qo: Str(Info),

if ( Is_Info_Str(p, Void, po) and Is_Info_Str(#p, Void, old_po) and Is_Info_Str(#q, Void, old_qo) )

then po ≤!≥ (old_po, old_qo) );
Let $rq\_shuffle$ be a perfect shuffle of info strings $r0$ and $q0$.

Then $pr \circ rq\_shuffle$ is a perfect shuffle of info strings $old\_p0$ and $old\_q0$. 
Loop Invariant

maintaining ( ∀ t: Location, if not Is_Reachable(#p, t) and not Is_Reachable(#q, t) then Target(t) = #Target(t) ) and ( ∀ pr, ro, qo, old_po, old_qo, rq_shuffle: Str(Info),
    if ( Is_Info_Str(p, r, po) and Is_Info_Str(r, Void, ro) and Is_Info_Str(q, Void, qo)
    Is_Info_Str(#p, Void, old_po) and Is_Info_Str(#q, Void, old_qo) and
    rq_shuffle ≤!≥ (ro, qo)
    then pr ◦ rq_shuffle ≤!≥ (old_po, old_qo) );
maintaining \( ( \forall t: \text{Location}, \text{if not Is\_Reachable}(\#p, t) \text{ and not Is\_Reachable}(\#q, t) \text{ then Target}(t) = \#\text{Target}(t) \text{ ) and} \)

\( ( \forall \text{pr, ro, qo, old\_po, old\_qo, rq\_shuffle: Str(Info)}, \text{if } ( \text{Is\_Info\_Str}(p, r, po) \text{ and Is\_Info\_Str}(r, Void, ro) \text{ and Is\_Info\_Str}(q, Void, qo) \text{ and Is\_Info\_Str}(\#p, Void, old\_po) \text{ and Is\_Info\_Str}(\#q, Void, old\_qo) \text{ and rq\_shuffle \leq !\geq (ro, qo)} \text{ then } \text{pr} \circ \text{rq\_shuffle} \leq !\geq (old\_po, old\_qo) ) ); \)
Sketch of Proof

**Initialization**
We need to show $pr \circ rq\_shuffle \leq !\geq (old\_po, old\_qo)$
Since $p = r$, it suffices to show
empty\_string $\circ pq\_shuffle \leq !\geq (old\_po, old\_qo)$

**Termination**
We know $pr \circ rq\_shuffle \leq !\geq (old\_po, old\_qo)$
Since $q = Void$, $rq\_shuffle = r0$, we know
$pr \circ ro \leq !\geq (old\_po, old\_qo)$
Sketch of Proof

Maintenance

Assume: \( pr \circ rq\_shuffle \leq ! \geq (old\_po, old\_qo) \)
Show: \( pr' \circ rq\_shuffle' \leq ! \geq (old\_po, old\_qo) \)

\( r_0 = \langle x \rangle \circ r_0' \)
\( q_0 = \langle y \rangle \circ q_0' \)

\( rq\_shuffle = \langle x \rangle \circ \langle y \rangle \circ rq\_shuffle' \)

\( ps' = pr \circ \langle x \rangle \)
\( pr' = pr \circ \langle x \rangle \circ \langle y \rangle \)

\( pr' \circ rq\_shuffle' = pr \circ rq\_shuffle \)

While (not Is\_At\_Void(q))

\[ \begin{align*}
\text{do} & \\
\text{Relocate}(s, r); & \\
\text{Follow\_Link}(r); & \\
\text{Redirect\_Link}(s, q); & \\
\text{Follow\_Link}(s); & \\
\text{Follow\_Link}(q); & \\
\text{Redirect\_Link}(s, r); & \\
\text{end;} & \\
\end{align*} \]
## Summary

<table>
<thead>
<tr>
<th>Traditional Analysis of Pointers</th>
<th>Specification-Based Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully automated</td>
<td>Partly automated</td>
</tr>
<tr>
<td>Relatively fast</td>
<td>Requires programmer supplied assertions</td>
</tr>
<tr>
<td>Low (but present) false positive rate</td>
<td>Handles both lightweight and heavyweight specifications</td>
</tr>
<tr>
<td>Limited in what it can prove</td>
<td></td>
</tr>
</tbody>
</table>
Related Work

- Region-based shape analysis
- Logic of stores
- Constraint solver with Alloy
- ESC/Java
- LOOP compiler
- Safe pointers and checked pointers
- Pointer component
Questions
Additional Slides
Objective

Allow programmers to reason about pointers and programs that involve pointers using the same techniques they use to reason about programs without pointers.

- The technique should not require special language semantics or proof rules that apply only to pointers.
- The technique should not be limited in what it can prove about programs with pointers.
Introduce a fully specified component into the programming language that captures the functional and performance behavior of pointers.

• Programmers can reason about pointers as they reason about any other component.

• For appropriate performance behavior, the compiler must implement the component differently than it implements other components.
Shape Analysis

Memory Abstraction

“Each configuration characterizes the state of one single heap location, called the **tracked location**.”
Swap_Contents($p, t$)
Shape Analysis

Overview

Reason about invariants that describe the “shapes” of dynamic data structures.

• A memory location is not referenced by more than one other location.
• A tree structure is maintained by a program.
• A list does not contain cycles.
• No accesses through dangling references.
• Memory leaks do not occur.
Shape Analysis: Tracking the State

08: List *z = y;
09: while (x != NULL)
10:   t = x;
11:   x = t->n;
12:   t->n = y->n;
13:   y->n = t;
14:   y = y->n->n;
16: return z;
Shape Analysis
Input state and abstraction

Example state at the beginning of splice
Shape Analysis

Output state and abstraction

Example state at the end of splice

Region
Points-to Component

Configuration
Shape Analysis

L2 presents a problem
Shape Analysis
L2 with acyclic output
Pointer Component
Redirect_Link$(p, q)$

before
After \( \alpha \rightarrow \beta \rightarrow \delta \) \(\text{Redirect\_Link}(p, q)\)
Splice Operation
**Definition Var** Is_Reachable_in (hops: \( \mathbb{N} \); p, q: Location): \( B \)

\[ = \text{Target}^{\text{hops}}(p) = q \text{ and } \forall k: \mathbb{N}, \text{if } \text{Target}^k(p) = q \text{ then } k \geq \text{hops}; \]

Is_Reachable_in(3, p, q)
Definition

**Definition Var**  \[ \text{Is\_Reachable} (p, q: \text{Location}): \mathbb{B} \]

\[= \exists k: \mathbb{N} \ni \text{Is\_Reachable\_in}(k, p, q); \]

\[\text{Is\_Reachable}(p, \text{Void})\]
Definitions

**Definition Var** Distance(p, q: Location): N

\[
Distance(p, q) = \begin{cases} 
  k & \text{if Is\_Reachable\_in}(k, p, q) \\
  0 & \text{otherwise}
\end{cases}
\]

Distance(q, Void) = 1
Definition Var \textbf{Is\_Info\_Str}(p, q: Location, \(\alpha\): Str(Info)): \(\mathcal{B}\)

\(\exists n: \mathbb{N} \ni \text{Is\_Reachable\_in}(n, p, q)\) and
\(\alpha = \prod_{k=1..n} \langle \text{Contents}(\text{Target}^k(p)) \rangle;\)

\textbf{Is\_Info\_Str}(p, q, \langle \alpha, \beta, \gamma \rangle)