Relational Inductive Shape Analysis

Simon Wegener
Green Slides

- Definitions
- Theorems
- Other formal stuff
Green Slides

- Definitions
- Theorems
- Other formal stuff

Yellow Slides

- Examples
Green Slides

- Definitions
- Theorems
- Other formal stuff

Yellow Slides

- Examples

Grey Slides

- Other stuff
- Some highlighted stuff
Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds∗
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O’Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a “separating conjunction” that asserts that its subformulas hold for disjoint parts of the heap, and a closely related “separating implication”. Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

1. Introduction

The use of shared mutable data structures, i.e., of structures where an updatable field can be referenced from more than one point, is widespread in areas as diverse as systems programming and artificial intelligence. Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size. (A partial bibliography is given in Reference [28].)

The problem faced by these approaches is that the correctness of a program that mutates data structures usually depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

\[ j := \text{nil} ; \text{while } i \neq \text{nil} \text{ do} \]
\[ (k := [i + 1] ; [i + 1] := j ; j := i ; i := k). \]

(Here the notation \([x] \) denotes the contents of the storage at address \(x\).)

The invariant of this program must state that \(i\) and \(j\) are lists representing two sequences \(\alpha\) and \(\beta\) such that the reflection of the initial value \(\alpha_0\) can be obtained by concatenating the reflection of \(\alpha\) onto \(\beta\):

\[ \exists \alpha, \beta. \text{list } \alpha \land \text{list } \beta \land \alpha_0 = \alpha^1 \cdot \beta, \]

where the predicate list \(\alpha\) is defined by induction on the length of \(\alpha\):

\[ \text{list } \epsilon \triangleq \text{nil} \]
\[ \text{list}(a \cdot \alpha) \triangleq \exists j. i \mapsto a, j \land \text{list } \alpha \]

(And \(\mapsto\) can be read as “points to”).

Unfortunately, however, this is not enough, since the program will malfunction if there is any sharing between the lists \(i\) and \(j\). To prohibit this we must extend the invariant to assert that only \(\text{nil}\) is reachable from both \(i\) and \(j\):

\[ \exists \alpha, \beta. \text{list } \alpha \land \text{list } \beta \land \alpha_0 = \alpha^1 \cdot \beta \]
\[ \land (\forall k. \text{reach}(i, k) \land \text{reach}(j, k) \Rightarrow k = \text{nil}), \]

where

\[ \text{reach}(i, j) \triangleq \exists n \geq 0. \text{reach}\_n(i, j) \]
\[ \text{reach}_0(i, j) \triangleq i = j \]
\[ \text{reach}_{n+1}(i, j) \triangleq \exists a, k. i \mapsto a, k \land \text{reach}_n(k, j). \]

Even worse, suppose there is some other list \(x\), representing a sequence \(\gamma\), that is not supposed to be affected by
Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O’Hearn and other early ideas of Burstall, we have developed a Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structures.

The simple imperative programming language attended to commands (not expressions) for modifying shared structures, and for explicit deallocation of storage. Assertions are extended using a “separating conjunction” that asserts formulas hold for disjoint parts of the heap, related “separating implication”. Coupled with a precise definition of predicates on abstract data structures, this extension permits the concise and flexible description of recursive procedures. We will also discuss promising future directions.

1. Introduction

The use of shared mutable data structures, where an updatable field can be referenced at more than one point, is widespread in areas as diverse as systems programming and artificial intelligence. Appropriate reasoning about this technique has been studied for three decades, but the result has been methods that suffer limited applicability or extreme complexity poorly to programs of even moderate size. (A history is given in Reference [28].)

The problem faced by these approaches is the rectness of a program that mutates data struc-

Shape Analysis with Structural Invariant Checkers*

Bor-Yuh Evan Chang 1, Xavier Rival 1,2, and George C. Necula 1

1 University of California, Berkeley, California, USA
2 École Normale Supérieure, Paris, France
{bec,rival,necula}@cs.berkeley.edu

Abstract. Developer-supplied data structure specifications are important to shape analyses, as they tell the analysis what information should be tracked in order to obtain the desired shape invariants. We observe that data structure checking code (e.g., used in testing or dynamic analysis) provides shape information that can also be used in static analysis. In this paper, we propose a lightweight, automatic shape analysis based on these developer-supplied structural invariant checkers. In particular, we set up a parametric abstract domain, which is instantiated with such checker specifications to summarize memory regions using both notions of complete and partial checker evaluations. The analysis then automatically derives a strategy for canonicalizing or weakening shape invariants.

1 Introduction

Pointer manipulation is fundamental in almost all software developed in imperative programming languages today. For this reason, verifying properties of interest to the developer or checking the pre-conditions for certain complex program transformations (e.g., refactoring) often requires detailed aliasing and structural information. Shape analyses are unique in that they can provide this detailed must-alias and shape information that is useful for many higher-level analyses (e.g., typestate or resource usage analyses, race detection for concurrent programs). Unfortunately, because of precision requirements, shape analyses have been generally prohibitively expensive to use in practice.

The design of our shape analysis is guided by the desire to keep the abstraction close to informal developer reasoning and to maintain a reasonable level of interaction with the user in order to avoid excessive case analysis. In this paper, we propose a shape analysis guided by the developer through programmer-supplied data structure invariants. The novel aspect of our proposal is that these
Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O'Hearn and other early ideas of Burstall, we have developed a Hoare logic that permits reasoning about low-level programs that use shared mutable data structures.

The simple imperative programming language extended with commands (not expressions) for modifying shared structures, and for explicit attention to the deletion of storage. Assertions are extended with a "separating conjunction" that asserts formulas hold for disjoint parts of the heap, related "separating implications". Coupled with the use of the separation of predicated data aspects permits the concise and flexible 2-structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that restricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising theoretical developments.

Shape Analysis with Structural Invariant Checkers∗

Bor-Yuh Evan Chang¹, Xavier Rival¹², and Georges Haddad³
¹ University of California, Berkeley, California, USA
² École Normale Supérieure, Paris, France
³ Département d'informatique, ENS Paris

Abstract. Developer-supplied data structure specifications are tant to shape analyses, as they tell the analysis what information can be tracked in order to obtain the desired shape invariants that data structure checking code (e.g., used in testing or debugging) provides shape information that can also be used in static program analysis. In this paper, we propose a lightweight, lightweight automatic shape analysis. We set up the parametric data structure analysis, which is instantiated with such specifications. In particular, we develop a lightweight, automatic shape analysis based on the use of generic shape checkers to summarize memory regions using complete and partial checker evaluations. The analysis typically derives a strategy for canonicalizing or weakening shape invariants.

1. Introduction

The use of shared mutable data structures, where an updateable field can be referenced by one point, is widespread in areas as diverse as systems and recursive procedures. We will also discuss promising theoretical developments.

1 Introduction

Pointer manipulation is fundamental in almost all software development languages today. For this reason, verifying shared memory is of interest to the developer or checking the pre-conditions for certain transformations (e.g., refactorings) often requires detailed analysis of shared memory. Shape analyses are unique in that they can track such information. Unfortunately, because of precision requirements, such analyses are generally expensive to use in practice.

The design of our shape analysis is guided by the desire to translate close to informal developer reasoning and to maintain a connection with the user in order to avoid excessive case analysis. In this paper, we propose a shape analysis guided by the developer that supersedes data structure invariants. The novel aspect of our proposal is that these inductive definitions can be checked for precision and can be obtained by simply dropping the constraint on the state of the checkers are constant across all recursive calls. This key component of shape analysis is a natural way for the developer to design an abstract domain that is parameterized both by an abstract domain for pure data properties and by user-supplied specifications of the data structure invariants to check. Particularly, it supports hybrid invariants about shape and data and features a general mechanism for combining shape analyses at the beginning, middle, or end of other analyses.

In this paper, we develop a lightweight shape analysis that tracks the state of the checkers as they are instantiations and design an abstract domain that is parameterized both by an abstract domain for pure data properties and by user-supplied specifications of the data structure invariants to check. Particularly, it supports hybrid invariants about shape and data and features a general mechanism for combining shape analyses at the beginning, middle, or end of other analyses. Around this domain, we build a shape analysis whose interesting components include a pre-analysis on the user-supplied specifications that guides the abstract interpretation and a widening operator over the combined shape and data domain. We then implement the shape analysis on the top of an assembler to the preservation of the red-black tree invariants during insertion.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programming]: Specifying and Verifying Abstract domains

Keywords shape analysis, inductive definitions, heap analysis, separation logic, symbolic abstract domain

Relational Inductive Shape Analysis

Bor-Yuh Evan Chang
University of California, Berkeley
bec@cs.berkeley.edu

Xavier Rival
INRIA and University of California, Berkeley
xavier@di.ens.fr

Abstract

Shape analyses are concerned with precise abstractions of the heap to capture detailed structural properties. To do so, they need to build and decompose summaries of disjoint memory regions. Unfortunately, many data structure invariants require relations that are tracked across disjoint regions, such as intricate numerical data invariants or structural invariants concerning back and cross pointers. In this paper, we identify issues inherent to analyzing relational structures and design an abstract domain that is parameterized both by an abstract domain for pure data properties and by user-supplied specifications of the data structure invariants to check. Particularly, it supports hybrid invariants about shape and data and features a general mechanism for combining shape analyses at the beginning, middle, or end of other analyses. Around this domain, we build a shape analysis whose interesting components include a pre-analysis on the user-supplied specifications that guides the abstract interpretation and a widening operator over the combined shape and data domain. We then implement the shape analysis on the top of a preservational assembler to the preservation of the red-black tree invariants during insertion.

1. Introduction

To obtain the necessary precision, shape analyses rely on inductive definitions for abstracting memory. In prior work (Chang et al., 2007), we proposed a shape analysis that combines inductive-defined predicates provided by the user. The novelty of our proposal is that these inductive definitions are not checked from checking code, that is, code that could be used to verify the invariants for the analysis. A nice property of using inductive defined predicates is that they are not only a familiar way for the developer to describe the data structure invariants but also express developer intent on how the data structure should be used.

In some respects, inductive-defined predicates are a natural fit for shape analysis (seemingly evidenced by the many shape analyses being built around them (Lee et al., 2005; Distefano et al., 2006; Rival et al., 2007; Magill et al., 2007; Gao et al., 2007). A key component of shape analysis is a materialization operation that enables strong updates, which are critical for precision. With an inductively-defined predicate, a natural materialization operation is to unfold its definition. For example, consider the following definition for an acyclic doubly-linked list:

1. \( \text{diff}(\text{prev}) \rightarrow \text{true} \) if \( \text{prev} = \text{false} \) then \( \text{false} \)

Here, we write inductive checkers in a pseudocode notation. The \( \text{diff} \) function defines a class of memory regions by a traversal from a designated root pointer (the traversal parameter). The \( \text{true} \) indicates the component corresponding to disjoint memory regions (i.e., the traversal is allowed to dereference each object field of a node). New variables (e.g., \( \text{prev} \)) are considered local variables to the body of the \( \text{diff} \) function.
Fast Facts

Introductory Examples

Background

Analysis

Summary
typedef struct TreeNode {
    struct TreeNode * l;
    struct TreeNode * r;
    struct TreeNode * p;
    int data;
} TreeNode;

void insert (TreeNode ** t, TreeNode * n) {
    assert(n != null && n->l == null && n->r == null && n->p == null);
    assert(isBST(t));
    TreeNode * p = null;
    TreeNode ** it = t;
    while (*it != null) {
        p = *it;
        if (n->data < p->data)
            it = &(p->l);
        else if (n->data > p->data)
            it = &(p->r);
        else
            return;
    }
    n->p = p;
    *it = n;
}
typedef struct TreeNode {
    struct TreeNode * l;
    struct TreeNode * r;
    struct TreeNode * p;
    int data;
} TreeNode;

void insert (TreeNode ** t, TreeNode * n) {
    assert (n != null && n->l == null && n->r == null && n->p == null);
    assert (isBST (t));
    TreeNode * p = null;
    TreeNode ** it = t;
    while (*it != null) {
        p = *it;
        if (n->data < p->data)
            it = &(p->l);
        else if (n->data > p->data)
            it = &(p->r);
        else
            return;
    }
    n->p = p;
    *it = n;
}

\[\langle \alpha, \beta, \delta \rangle := \langle \alpha = \text{null} \rangle \]

Fast Facts
Fast Facts

Source Code

\[ \alpha \text{dll}(\emptyset) := \langle \text{emp}, \{ \alpha = \text{null} \} \rangle \]

\[ \forall t : \beta \rightarrow \alpha, \{ \alpha \rightarrow \text{prev} \rightarrow \delta \rightarrow \beta \cdot \phi \} \]
Source Code

data (emp, {α = null})

Data Lattice

Fast Facts
Fast Facts

- Source Code
- Data Lattice
- Invariant Specification
Fast Facts
Fast Facts

3-valued logic (e.g. TVLA)

separation logic
Example: Singly-Linked List
Example: Singly-Linked List

1.list := if (l = null) then [] else l@next ↦ n
  * n.list
Example: Singly-Linked List

```plaintext
l.list := if (l = null) then [] else l@next ↦ n * n.list
```

empty memory region
Example: Singly-Linked List

\[
\begin{align*}
\text{l.list := } & \text{ if } (l = \text{null)} \\
\text{then } & [] \\
\text{else } & l@\text{next} \mapsto n \\
& \ast n.\text{list}
\end{align*}
\]

address where \( l \) points to plus offset of field \( next \)
Example: Singly-Linked List

l.list := if (l = null)
then []
else l@next ↦ n
* n.list

address $l@next$ points to $n$
Example: Singly-Linked List

```plaintext
l.list := if (l = null)
then []
else l@next ↦ n
   * n.list
```

$n$ symbolic variable
Example: Singly-Linked List

l.list := if (l = null) then [] else l@next ↦ n

* n.list

n has to be a list again
l.list := \textbf{if} (l = \texttt{null})
\textbf{then} []
\textbf{else} l@\texttt{next} \mapsto n

* n.list

* \texttt{(star)} separates two memory regions
Example: Singly-Linked List

\[ l = \begin{cases} [\ ] & \text{if } (l = \text{null}) \\ l@\text{next} \mapsto n & \text{else } \end{cases} \quad (n = 1, 2, 3, 4) \]

\[ l.\text{list} := \begin{cases} \textbf{if} (l = \text{null}) \\ \text{then } [\ ] \\ \text{else } l@\text{next} \mapsto n & \text{else } \end{cases} \quad (n = 1, 2, 3, 4) \]

* n.list
Example: Singly-Linked List

l\text{.list} := \text{if } (l = \text{null}) \text{ then } [] \text{ else } l\text{.@next} \mapsto n

* n\text{.list}
l.list := if (l = null)
then []
else l@next ↦ n
    * n.list
Example: Singly-Linked List

\[
\text{l.list := if } (\text{l} = \text{null}) \text{ then } [] \text{ else } \text{l@next} \mapsto n \quad \text{* n.list}
\]

\text{no relation between nodes}
Example: Doubly-Linked List

1  2  3  4
null  null

data
next
prev
Example: Doubly-Linked List

\[ l.dll (l_{prev}) := \textbf{if} \ (l = \text{null}) \]
\[ \text{then} \ [] \]
\[ \text{else} \ l@\text{next} \mapsto n \]
\[ * \ l@\text{prev} \mapsto l_{prev} \]
\[ * \ n.dll(l) \]
Example: Doubly-Linked List

\[
\begin{align*}
\text{l.dll (l\_prev)} & := \begin{cases} 
\text[]{} & \text{if } (l = \text{null}) \\
\text{l@next } & \mapsto \text{n} \\
\ast \, \text{l@prev } & \mapsto \text{l\_prev} \\
\ast \, \text{\text{n.dll(l)} & \end{cases} 
\end{align*}
\]

\text{invertible structural invariant:}
following next and then prev brings you back to the same node
Example: Binary Search Tree
Example: Binary Search Tree

\[
t.\text{bst} (t_{lo}, t_{hi}) := \begin{cases} 
\text{[]} & \text{if } (t = \text{null}) \\
\text{t@l} \mapsto l & \text{and } t_{lo} < t_{hi} \\
\text{t@r} \mapsto r & \text{and } t_{lo} < d < t_{hi} \\
\text{t@data} \mapsto d & \text{and } t_{lo} < d < t_{hi} \\
\text{l.bst}(t_{lo}, d) & \text{and } t_{lo} < d < t_{hi} \\
\text{r.bst}(d, t_{hi}) & \text{and } t_{lo} < d < t_{hi} 
\end{cases}
\]
Example: Binary Search Tree

t bst (t lo, t hi) := if (l = null) then [] and t lo < t hi
else t @ l ↦ l
    * t @ r ↦ r
    * t @ data ↦ d
    * l bst(t lo, d)
    * r bst(d, t hi)
and t lo < d < t hi

the shape constrains the data
✓ Fast Facts

✓ Introductory Examples

▶ Background

Analysis

Summary
Separation Logic

Values = Integers

Addr \subseteq Values

stack: Var \rightarrow Values

heap: Addr \rightarrow Values

h_0 \perp h_1: \text{heaps } h_0 \text{ and } h_1 \text{ have disjoint domains}

h_0 \cdot h_1: \text{union of such heaps}
\[ \langle \text{assert} \rangle ::= \text{usual logical connectives} \]

\[ \begin{align*}
| \quad \text{emp} & \quad \text{(empty heap)} \\
| \quad \langle \text{exp} \rangle \leftrightarrow \langle \text{exp} \rangle & \quad \text{(singleton heap)} \\
| \quad \langle \text{assert} \rangle \ast \langle \text{assert} \rangle & \quad \text{(separating conjunction)} \\
| \quad \langle \text{assert} \rangle -\ast \langle \text{assert} \rangle & \quad \text{(separating implication)}
\end{align*} \]

\[ s, h \vdash p \text{ iff stack } s \text{ and heap } h \text{ conform to assertion } p \]
Separation Logic

\[
\begin{array}{c}
x \rightarrow 3 \\
\end{array}
\]

\[
\begin{array}{c}
s, h \vdash x \leftrightarrow 3 \\
s, h \vdash true \\
s, h \not\vdash x \leftrightarrow 4 \\
s, h \not\vdash emp \\
\end{array}
\]
Separation Logic

\[ s, h \vdash x \mapsto 3 \times x + 1 \mapsto y \]
\[ s, h \vdash y \mapsto 3 \times y + 1 \mapsto x \]

\[ s, h \vdash (x \mapsto 3 \times x + 1 \mapsto y) \times (y \mapsto 3 \times y + 1 \mapsto x) \]
Separation Logic

\[ s, h \vdash x \mapsto 3 \ast x + 1 \mapsto y \]

\[ s, h \vdash y \mapsto 3 \ast y + 1 \mapsto x \]

\[ s, h \vdash (x \mapsto 3 \ast x + 1 \mapsto y) \text{ and } (y \mapsto 3 \ast y + 1 \mapsto x) \]
s, h ⊨ p_0 * p_1 \text{ iff } \\
\exists h_0, h_1. h_0 \perp h_1 \text{ and } h = h_0 \cdot h_1 \text{ and } s, h_0 \vdash p_0 \text{ and } s, h_1 \vdash p_1
Separating Implication

\[ s, h \vdash p_0 \not\leftrightarrow p_1 \text{ iff } \forall h_0. (h_0 \perp h \text{ and } s, h_0 \vdash p_0) \text{ implies } s, (h_0 \cdot h) \vdash p_1 \]
Separating Implication

\[ s, h \vdash p_0 \leftrightarrow p_1 \text{ iff } \forall h_0. (h_0 \perp h \text{ and } s, h_0 \vdash p_0) \text{ implies } s, (h_0 \cdot h) \vdash p_1 \]

\[ x \rightarrow 3 \]

\[ s, h \vdash y \leftrightarrow 5 \leftrightarrow (x \leftrightarrow 3 \leftrightarrow y \leftrightarrow 5) \]

\[ x \rightarrow 3 \quad y \rightarrow 5 \]

\[ x \rightarrow 3 \quad y \rightarrow 5 \]

\[ s, h_0 \cdot h \vdash x \leftrightarrow 3 \leftrightarrow y \leftrightarrow 5 \]
✓ Fast Facts
✓ Introductory Examples
✓ Background
  ▶ Analysis
  
Summary
Abstract Interpretation
Abstraction
Abstraction
Analysis State

\[ A ::= \bot \mid \langle E, M, P \rangle \mid A_1 \lor A_2 \]
\[ E ::= \cdot \mid E, x \mapsto \alpha \quad \text{(environment)} \]
\[ P \in P^# \quad \text{(data constraints)} \]
\[ \alpha \in \text{Val}^# \quad \text{(symbolic values)} \]
\[ x \in \text{Var} \quad \text{(program variables)} \]
\( M ::= \text{emp} \quad \text{(empty)} \)

\[
| \ A \ @ \ f \rightarrow \ B \\
| \ M_0 \ast M_1 \quad \text{(disjoint regions)} \\
| \ A.c(\delta) \quad \text{(checker region)} \\
| \ A.c(\delta) \ast= A'.c'(\delta') \quad \text{(checker segment)}
\]
Memory Abstraction

\[ \alpha @ f \mapsto \beta \]

\[ \alpha.c(\delta) \]

\[ \alpha.c(\delta) \ast= \alpha'.c'(\delta') \]
t.bst (t_{lo}, t_{hi}) := \textbf{if} (l = \text{null})\textbf{ then} [] \textbf{ and } t_{lo} < t_{hi} \textbf{ else} t[l] \mapsto l \\
\ast t[r] \mapsto r \\
\ast t[data] \mapsto d \\
\ast l.bst(t_{lo}, d) \\
\ast r.bst(d, t_{hi}) \\
\textbf{and} t_{lo} < d < t_{hi}
Inductive Checker Definitions

\[ \alpha.bst(\delta_{lo}, \delta_{hi}) := \]
\[ \langle \text{emp}, \{\alpha = \text{null}, \delta_{lo} < \delta_{hi}\} \rangle \]
\[ \lor \]
\[ \langle \alpha@l \mapsto \beta \]
\[ * \alpha@r \mapsto \gamma \]
\[ * \alpha@data \mapsto \alpha_d \]
\[ * \beta.bst(\delta_{lo}, \alpha_d) \]
\[ * \gamma.bst(\alpha_d, \delta_{hi}), \]
\[ \{\delta_{lo} < \alpha_d < \delta_{hi}\} \rangle \]

\[ \text{t.bst}(t_{lo}, t_{hi}) := \text{if } (l = \text{null}) \]
\[ \quad \text{then } [] \text{ and } t_{lo} < t_{hi} \]
\[ \quad \text{else } t@l \mapsto l \]
\[ \quad * t@r \mapsto r \]
\[ \quad * t@data \mapsto d \]
\[ \quad * l.bst(t_{lo}, d) \]
\[ \quad * r.bst(d, t_{hi}) \]
\[ \text{and } t_{lo} < d < t_{hi} \]
Inductive Segments

\[ \alpha.c(\delta) \ast- \alpha'.c'(\delta') \]
useful for summarizing segments,
but does not allow unfolding
\[ \Rightarrow \]
\[ \alpha.c(\delta) \ast= \alpha'.c'(\delta') \]
\[ \alpha.c(\delta) \ast= \alpha'.c'(\delta') := \]

M, M´ disjoint memory regions, 
M´ satisfies \( \alpha'.c'(\delta') \)
\[ \Rightarrow \]
M \ast M´ satisfies \( \alpha.c(\delta) \) up to some number of unfoldings.
Inductive Segments

\[ \alpha.c(\delta) \ast= \alpha'.c'(\delta') := \]

\( M, M' \) disjoint memory regions,

\( M' \) satisfies \( \alpha'.c'(\delta') \)

\[ \Rightarrow \]

\( M * M' \) satisfies \( \alpha.c(\delta) \) up to some number of unfoldings.
Working in the Abstract Domain
Updates in the Shape Domain
x→l = null;
Updates in the Shape Domain

\[ x \rightarrow l = \text{null}; \]
Updates in the Shape Domain

\[
x \rightarrow l = \text{null};
\]

\[
x \rightarrow r = y;
\]
Updates in the Shape Domain

\[ x \rightarrow l = \text{null}; \]

\[ x \rightarrow r = y; \]
Updates in the Shape Domain

\[
x \rightarrow l = \text{null}; \\
x \rightarrow r = y;
\]

\[\Rightarrow \text{needs to materialize (i.e. partial concretize)}\]
Inductive Checker Definitions

\[ \alpha.\text{bst} (\delta_{lo}, \delta_{hi}) := \]

\[ \langle \text{emp}, \{\alpha = \text{null}, \delta_{lo} < \delta_{hi}\} \rangle \lor \langle \alpha_@l \mapsto \beta \]

\[ \quad \ast \alpha_@r \mapsto \gamma \]

\[ \quad \ast \alpha_@\text{data} \mapsto \alpha_\text{d} \]

\[ \quad \ast \beta.\text{bst}(\delta_{lo}, \alpha_\text{d}) \]

\[ \quad \ast \gamma.\text{bst}(\alpha_\text{d}, \delta_{hi}) , \{\delta_{lo} < \alpha_\text{d} < \delta_{hi}\} \rangle \]}
Unfolding of Inductive Checkers

\( \xi \xrightarrow{\text{bst}(\xi_{lo}, \xi_{hi})} x \)

\[ \neg \exists \text{emp} \quad \xi = \text{null} \land \xi_{lo} < \xi_{hi} \]

\[ \neg \exists \text{null} \land \xi_{lo} < \xi_{d} < \xi_{hi} \]

\( \xi \xrightarrow{\text{bst}(\xi_{lo}, \xi_{d})} \text{bst}(\xi_{d}, \xi_{hi}) \)

\( \xi \xrightarrow{\text{bst}(\xi_{lo}, \xi_{hi})} \text{bst}(\xi_{d}, \xi_{hi}) \)

\( \xi \xrightarrow{\text{@data} \Rightarrow \xi_{d}} \text{bst}(\xi_{lo}, \xi_{d}) \)

\( \xi \xrightarrow{r} \text{bst}(\xi_{d}, \xi_{hi}) \)
Forward Unfolding of Inductive Segments

\[\text{\texttt{bst(lo, hi)}} \Rightarrow \text{\texttt{emp}}\]

\[\text{\texttt{bst(lo, hi)}} \Rightarrow \text{\texttt{bst(lo, hi)}}\]

\[\text{\texttt{P}} \land \xi \neq \text{\texttt{null}} \land \xi_{lo} < \xi_{d} < \xi_{hi}\]

\[\text{\texttt{P} \land \xi = \text{\texttt{null}} \land \xi_{lo} < \xi_{hi} \land \xi_{lo} = \eta_{lo} \land \xi_{hi} = \eta_{hi} \land \xi = \eta}\]
Backwards Traversal

\[ \alpha . dll(\delta_{\text{prev}}) := \]

\[ \langle \text{emp}, \{\alpha = \text{null}\} \rangle \quad \lor \quad \langle \alpha \@ \text{next} \mapsto \beta \quad \ast \quad \alpha \@ \text{prev} \mapsto \delta_{\text{prev}} \quad \ast \quad \beta . dll(\alpha), \{\}\rangle \]
Backward Unfolding of Inductive Segments
We can split inductive checker segments!

\[(s, h) \in \gamma (\alpha.c(\delta) \ast=i+1 \alpha'.c'(\delta')) \Rightarrow \]

\[(s, h) \in \gamma (\alpha.c(\delta) \ast=i \alpha''.c''.(\delta'') \ast \alpha''.c'(\delta'')) \ast=1 \alpha'.c'(\delta')) \]
Backward Unfolding of Inductive Segments

\[
\alpha \xrightarrow{\text{dll(null)}} \beta \xrightarrow{\text{prev}} \gamma
\]
Backward Unfolding of Inductive Segments

\[ \alpha \xrightarrow{\text{dll(null)}} \beta' \xrightarrow{\text{dll(\(\xi\)}} \gamma \]

Nodes: \(\alpha, \beta', \gamma\)

Edges: \(\alpha \xrightarrow{1} \beta', \beta' \xrightarrow{\text{dll(\(\xi\)}} \gamma, \beta' \xrightarrow{\text{dll(\(\xi\)}} \gamma\)

Labels: dll(null), dll(\(\xi\)), dll(\(\xi\)), dll(\(\beta\))
Backward Unfolding of Inductive Segments
Backward Unfolding of Inductive Segments
Working in the Abstract Domain, cont’d
Approximation Test

less precise

more precise
Approximation Test

\[
\gamma_0 \xrightarrow{\text{bst}(\gamma_0 \text{ lo}, \gamma_0 \text{ hi})} \gamma_1 \quad \text{less precise}
\]

\[
\beta_0 \xrightarrow{\text{bst}(\beta_0 \text{ d}, \beta_0 \text{ hi})} \beta_2 \quad \text{more precise}
\]
Approximation Test

\[ \gamma_0 \sim \beta_0 \]

\[ \gamma_0 \rightarrow \beta_0 \]

Diagram:

- Two paths from \( \gamma_0 \) to \( \gamma_1 \):
  - \( \text{bst}(\gamma_0 \text{ lo}, \gamma_0 \text{ hi}) \)
  - \( \text{bst}(\gamma_1 \text{ lo}, \gamma_1 \text{ hi}) \)

- Two paths from \( \beta_0 \) to \( \beta_1 \):
  - \( \text{bst}(\beta_0 \text{ lo}, \beta_0 \text{ hi}) \)

- Two edges from \( \gamma_1 \) to \( \gamma_2 \):

- Two edges from \( \beta_1 \) to \( \beta_2 \):
  - \( \text{bst}(\beta_0 \text{ lo}, \beta_0 \text{ hi}) \)
  - \( \text{bst}(\beta_0 \text{ lo}, \beta_0 \text{ hi}) \)

- Two edges from \( \beta_2 \) to \( \beta_3 \):
  - \( \text{bst}(\beta_0 \text{ lo}, \beta_0 \text{ hi}) \)
  - \( \text{bst}(\beta_0 \text{ lo}, \beta_0 \text{ hi}) \)
Approximation Test

\[ \gamma_0 \rightarrow \beta_0 \]
\[ \gamma_1 \rightarrow \beta_1 \]
Approximation Test

\[ \gamma_0 \sim \beta_0 \]
\[ \gamma_1 \sim \beta_1 \]
Approximation Test

\[\gamma_0 \sim \beta_0\]

\[\gamma_1 \sim \beta_1\]

\[\gamma_0 \neq \text{null} \land \gamma_0 \text{lo} < \gamma_0 \text{d} < \gamma_0 \text{hi}\]
Approximation Test

\[ \gamma_0 \rightarrow \beta_0 \]
\[ \gamma_1 \rightarrow \beta_1 \]
\[ \gamma_1' \rightarrow \beta_1 \]
\[ \gamma_2 \rightarrow \beta_2 \]
\[ \gamma_0 \text{ lo} \rightarrow \beta_0 \text{ lo} \]
\[ \gamma_0 \text{ d} \rightarrow \beta_0 \text{ d} \]
\[ \gamma_0 \text{ hi} \rightarrow \beta_0 \text{ hi} \]
Approximation Test

\[ \gamma_1' \xrightarrow{\text{bst}(\gamma_0 \text{ lo}, \gamma_0 \text{ d})} \beta_0 \]
\[ \gamma_1 \xrightarrow{\text{bst}(\gamma_1 \text{ lo}, \gamma_1 \text{ hi})} \beta_1 \]
\[ \gamma_0 \rightarrow \beta_0 \]
\[ \gamma_1 \rightarrow \beta_1 \]
\[ \gamma_1' \rightarrow \beta_1 \]
\[ \gamma_2 \rightarrow \beta_2 \]
\[ \gamma_0 \text{ lo} \rightarrow \beta_0 \text{ lo} \]
\[ \gamma_0 \text{ d} \rightarrow \beta_0 \text{ d} \]
\[ \gamma_0 \text{ hi} \rightarrow \beta_0 \text{ hi} \]
Approximation Test

\[ \gamma_1' \xrightarrow{\text{bst}(\gamma_0 \text{ lo}, \gamma_0 \text{ d})} \gamma_0 \xrightarrow{\text{bst}(\gamma_1 \text{ lo}, \gamma_1 \text{ hi})} \gamma_1 \]

\[ \begin{align*}
\gamma_0 & \rightarrow \beta_0 \\
\gamma_1 & \rightarrow \beta_1 \\
\gamma_1' & \rightarrow \beta_1 \\
\gamma_2 & \rightarrow \beta_2 \\
\gamma_0 \text{ lo} & \rightarrow \beta_0 \text{ lo} \\
\gamma_0 \text{ d} & \rightarrow \beta_0 \text{ d} \\
\gamma_0 \text{ hi} & \rightarrow \beta_0 \text{ hi} \\
\gamma_1 \text{ lo} & \rightarrow \beta_0 \text{ lo} \\
\gamma_1 \text{ hi} & \rightarrow \beta_0 \text{ d}
\end{align*} \]
Approximation Test

\[
\begin{align*}
\gamma_0 \rightarrow \beta_0 \\
\gamma_1 \rightarrow \beta_1 \\
\gamma_1' \rightarrow \beta_1 \\
\gamma_2 \rightarrow \beta_2 \\
\gamma_0 \text{ lo} \rightarrow \beta_0 \text{ lo} \\
\gamma_0 \text{ d} \rightarrow \beta_0 \text{ d} \\
\gamma_0 \text{ hi} \rightarrow \beta_0 \text{ hi} \\
\gamma_1 \text{ lo} \rightarrow \beta_0 \text{ lo} \\
\gamma_1 \text{ hi} \rightarrow \beta_0 \text{ d}
\end{align*}
\]
Join

$M_0 \sqcup M_1$
Join

$M'$

$M_0 \sqcup M_1$
A widening operator $\nabla$ is a join operator with a stabilizing property to ensure termination.

$\sqcap$ is a widening operator.
Concretization
$\gamma(M) = \{ (s, h) \mid s, h \vdash M \}$
Concretization

\[ \zeta \xrightarrow{\text{bst}(0, 6)} t \]
Concretization

bst(0, 6)
Concretization

\[ \zeta \xrightarrow{\text{bst}(0, 6)} \]

\[ t \xrightarrow{\text{null}} \]

\[ 1 \xrightarrow{\text{null}} \]

\[ 2 \xrightarrow{\text{null}} \]

\[ 3 \xrightarrow{\text{null}} \]

\[ 5 \xrightarrow{\text{null}} \]

\[ 4 \xrightarrow{\text{null}} \]
Concretization
Concretization

\[ \xi \rightarrow \text{bst}(0, 6) \]

\[ \begin{array}{c}
4 \\
5 \\
2 \\
1 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
4 \\
5 \\
2 \\
1 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
3 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
3 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
1 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
1 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
5 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
5 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
4 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
4 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \rightarrow \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \]

\[ \begin{array}{c}
2 \\
\null \\
\null \\
\null
\end{array} \]
✓ Fast Facts
✓ Introductory Examples
✓ Background
✓ Analysis
› Summary
What have we seen?
t bst (t lo, t hi) := if (l = null) 
then [] and t lo < t hi
else t@l = l
  * t@r = r
  * t@data = d
  * l.bst(t lo, d)
  * r.bst(d, t hi)
and t lo < d < t hi

What have we seen?
What have we seen?

Inductive Checker Definitions

Memory Abstraction

M ::= emp (empty)
| α @ f ← β (memory cell)
| M₀ * M₁ (disjoint regions)
| α.c(δ) (checker region)
| α.c(δ) * α′.c′(δ′) (checker segment)
What have we seen?
What have we seen?

Inductive Checker Definitions

Memory Abstraction

Working in the Abstract Domain

Unfolding of Inductive Checkers

\[ t_{bst}(t_{lo}, t_{hi}) := \]
\[ \begin{cases} \text{if } (l = \text{null}) \text{ then } & \emptyset \text{ and } t_{lo} < t_{hi} \text{ else } \end{cases} \]
\[ t_{@}l \text{! } l \text{! } t_{@}r \text{! } r \text{! } t_{@data} \text{! } d \]
\[ l_{bst}(t_{lo}, d) \text{! } r_{bst}(d, t_{hi}) \]
\[ \text{and } t_{lo} < d < t_{hi} \]

Inductive Checker Definitions

Memory Abstraction

Working in the Abstract Domain

Unfolding of Inductive Checkers

\[ M ::= \]
\[ M_0 \]
\[ M_1 \]
\[ .c(\#) \]
\[ .c(\#) = \cdot.c(\#') \]

Working in the Abstract Domain

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]

Unfolding of Inductive Checkers

\[ \text{emp} \]
\[ x \]
\[ \text{bst}(\zeta_{lo}, \zeta_{hi}) \]
\[ \zeta = \text{null } \land \zeta_{lo} < \zeta_{hi} \]
What to stress?

- **User-supplied** shape *invariants* and data domains
- Various *relations* between shape and data can be tracked
- **Arbitrary data structures** can be analyzed
- **Backward unfolding** of checker segments through forward unfolding
Questions ?