Monotonic Abstraction for Programs with Muliply-Linked Structures

Timotheus Jochum
Supervision: David Korzeniewski

Rheinisch-Westfaelische Technische Hochschule Aachen - Bc. Informatik

14. Februar 2017

Introduction

Introduction

- Verification of programs

Introduction

- Verification of programs
- Programs written in a subset of C

Introduction

- Verification of programs
- Programs written in a subset of C
- Operating on multiply linked data structures

Outline

Outline

- Introduction

Outline

- Introduction
- Structures

Outline

- Introduction
- Structures
- Heap

Outline

- Introduction
- Structures
- Heap
- Signatures

Outline

- Introduction
- Structures
- Heap
- Signatures
- Monotonic Abstraction

Outline

- Introduction
- Structures
- Heap
- Signatures
- Monotonic Abstraction
- Computing Predecessors

Outline

- Introduction
- Structures
- Heap
- Signatures
- Monotonic Abstraction
- Computing Predecessors
- Reachability Algorithm

Outline

- Introduction
- Structures
- Heap
- Signatures
- Monotonic Abstraction
- Computing Predecessors
- Reachability Algorithm
- Conclusion

Structures

Structures

- C structs with two selectors

Structures

- C structs with two selectors

```
struct A {
    A *next;
    A *prev;
    Data data;
}
```

Structures

- C structs with two selectors

```
struct A {
    A *next;
    A *prev;
    Data data;
}
```

- Allows complex structs

Structures

- C structs with two selectors

```
struct A {
    A *next;
    A *prev;
    Data data;
}
```

- Allows complex structs

```
struct B {
    T1 t1;
    T2 t2;
    T3 t3;
    T4 t4;
}
```

Heap

Heap

- Vertex and edge labeled graph

Heap

- Vertex and edge labeled graph
- Each node represents a memory cell

Heap

- Vertex and edge labeled graph
- Each node represents a memory cell
- Each edge represents a pointer/selector

Heap

- Vertex and edge labeled graph
- Each node represents a memory cell
- Each edge represents a pointer/selector
- '#' represents the *null node*

Heap

- Vertex and edge labeled graph
- Each node represents a memory cell
- Each edge represents a pointer/selector
- '#' represents the *null node*
- '*' represents the *dangling node*

Heap

- Vertex and edge labeled graph
- Each node represents a memory cell
- Each edge represents a pointer/selector
- '#' represents the *null node*
- '*' represents the *dangling node*
- Labels on nodes for variables

Heap

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

- $\overline{M} = M \cup \{\#, *\}$, where M is the set of memory cells

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

- $\overline{M} = M \cup \{\#, *\}$, where M is the set of memory cells
- E is the set of edges

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

- $\overline{M} = M \cup \{\#, *\}$, where M is the set of memory cells
- E is the set of edges
- $s : E \to M$ is the *source function*

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

- $\overline{M} = M \cup \{\#, *\}$, where M is the set of memory cells
- E is the set of edges
- $s : E \to M$ is the *source function*
- $t : E \to \overline{M}$ is the *target function*

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

- $\overline{M} = M \cup \{\#, *\}$, where M is the set of memory cells
- E is the set of edges
- $s : E \to M$ is the *source function*
- $t : E \to \overline{M}$ is the *target function*
- $\tau : E \to C$ is the *type function*, where $C = \{1, 2\}$ is the set of selectors

Heap

Definition

A *heap* is defined as a tuple $(\overline{M}, E, s, t, \tau, \lambda)$ where:

- $\overline{M} = M \cup \{\#, *\}$, where M is the set of memory cells
- E is the set of edges
- $s : E \to M$ is the *source function*
- $t : E \to \overline{M}$ is the *target function*
- $\tau : E \to C$ is the *type function*, where $C = \{1, 2\}$ is the set of selectors
- $\lambda : X \to \overline{M}$ is the *variable function*, where X is a set of variables

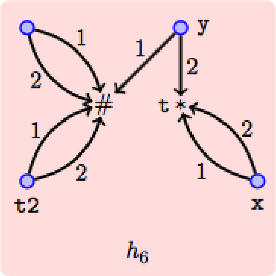
Heap

Heap

Example of a heap:

Heap

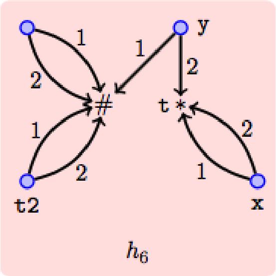
Example of a heap:



Example Heap [ACV13]

Heap

Example of a heap:

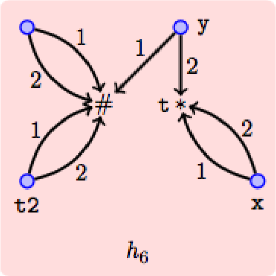


Example Heap [ACV13]

- The heap invariant:

Heap

Example of a heap:



Example Heap [ACV13]

- The heap invariant:
- $\forall c \in C\ \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| = 1$

Heap Transition System

Heap Transition System

- Subset of C includes the following operations:

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x \mathrel{!}= y$

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x \mathrel{!}= y$
  - Assignments: $x = y$, $x = y.next(i)$, $x.next(i) = y$

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x \mathrel{!}= y$
  - Assignments: $x = y$, $x = y.next(i)$, $x.next(i) = y$
  - Allocation: $x = malloc()$ and $free(x)$

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x \mathrel{!}= y$
  - Assignments: $x = y$, $x = y.next(i)$, $x.next(i) = y$
  - Allocation: $x = malloc()$ and $free(x)$
- The Transition System is defined as $T = (S, \rightarrow)$

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x \mathrel{!}= y$
  - Assignments: $x = y$, $x = y.next(i)$, $x.next(i) = y$
  - Allocation: $x = malloc()$ and $free(x)$
- The Transition System is defined as $T = (S, \rightarrow)$
  - S is a set of states which are pairs of (pc, h) where pc is the Program Counter

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x\ != y$
  - Assignments: $x = y$, $x = y.next(i)$, $x.next(i) = y$
  - Allocation: $x = malloc()$ and $free(x)$
- The Transition System is defined as $T = (S, \rightarrow)$
  - S is a set of states which are pairs of (pc, h) where pc is the Program Counter
  - $\rightarrow$ reflects the heap manipulation

Heap Transition System

- Subset of C includes the following operations:
  - Conditions: $x == y$, $x \mathrel{!=} y$
  - Assignments: $x = y$, $x = y.next(i)$, $x.next(i) = y$
  - Allocation: $x = malloc()$ and $free(x)$
- The Transition System is defined as $T = (S, \rightarrow)$
  - S is a set of states which are pairs of (pc, h) where pc is the Program Counter
  - $\rightarrow$ reflects the heap manipulation
- Given states s, s', $s \rightarrow s'$ exists if an operation is defined which transforms h into h'

Signatures

Signatures

- Defined in the same way as heaps except for:

Signatures

- Defined in the same way as heaps except for:
- The *type function* and the *variable function* can be partial

Signatures

- Defined in the same way as heaps except for:
- The *type function* and the *variable function* can be partial
- Less strict invariants:

Signatures

- Defined in the same way as heaps except for:
- The *type function* and the *variable function* can be partial
- Less strict invariants:
  - $\forall c \in C \; \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| \leq 1$

Signatures

- Defined in the same way as heaps except for:
- The *type function* and the *variable function* can be partial
- Less strict invariants:
  - $\forall c \in C \; \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| \leq 1$
  - $\forall m \in M : |s^{-1}(m)| \leq 2$

Signatures

- Defined in the same way as heaps except for:
- The *type function* and the *variable function* can be partial
- Less strict invariants:
  - $\forall c \in C \; \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| \leq 1$
  - $\forall m \in M : |s^{-1}(m)| \leq 2$
- Signatures are partial heaps

Signatures

- Defined in the same way as heaps except for:
- The *type function* and the *variable function* can be partial
- Less strict invariants:
  - $\forall c \in C \ \forall m \in M : |s^{-1}(m) \cap \tau^{-1}(c)| \leq 1$
  - $\forall m \in M : |s^{-1}(m)| \leq 2$
- Signatures are partial heaps
- What does a Signature actually represent?

Signatures

Signatures

- Signatures represent infinite sets of heaps

Signatures

- Signatures represent infinite sets of heaps
- Or in other words:

Signatures

- Signatures represent infinite sets of heaps
- Or in other words:

A signature is a heap with some parts 'missing'. It represents the set of all heaps that have at least the property or structural information described by this signature.

Ordering on Signatures

Ordering on Signatures

Ordering Steps:

Ordering on Signatures

Ordering Steps:

- Isolated cell deletion

Ordering on Signatures

Ordering Steps:

- Isolated cell deletion
- Edge deletion

Ordering on Signatures

Ordering Steps:

- Isolated cell deletion
- Edge deletion
- Contraction

Ordering on Signatures

Ordering Steps:

- Isolated cell deletion
- Edge deletion
- Contraction
- Edge decoloring

Ordering on Signatures

Ordering Steps:

- Isolated cell deletion
- Edge deletion
- Contraction
- Edge decoloring
- Label deletion

The Ordering Relation

The Ordering Relation

Definition

We say a signature sig_1 is smaller than a signature sig_2, if there is a sequence of ordering steps that leads from sig_2 to sig_1, written as $sig_1 \sqsubseteq sig_2$.

The Ordering Relation

Definition

We say a signature sig_1 is smaller than a signature sig_2, if there is a sequence of ordering steps that leads from sig_2 to sig_1, written as $sig_1 \sqsubseteq sig_2$.

- We say that a heap h satisfies sig_1, written as $h \in [\![sig_1]\!]$, if $sig_1 \sqsubseteq h$

Bad Configurations

Bad Configurations

- Heap Configurations that should **never** occur

Bad Configurations

- Heap Configurations that should **never** occur
- Represented as signatures

Bad Configurations

- Heap Configurations that should **never** occur
- Represented as signatures
- Example of a Bad Configuration:

Bad Configurations

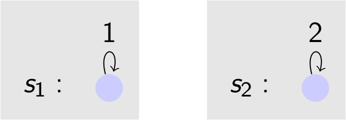
- Heap Configurations that should **never** occur
- Represented as signatures
- Example of a Bad Configuration:

Non-Cyclicity: This bad configuration refers to all structures which have a selector pointing to the same memory cell m.

Bad Configurations

- Heap Configurations that should **never** occur
- Represented as signatures
- Example of a Bad Configuration:

Non-Cyclicity: This bad configuration refers to all structures which have a selector pointing to the same memory cell m.



Monotonic Abstraction

Monotonic Abstraction

Definition

A transition system T is monotonic if the following holds. For any states sig_1, sig_2 and sig_3 such that $sig_1 \sqsubseteq sig_2$ and $sig_1 \longrightarrow sig_3$, we can always find a state sig_4 such that $sig_2 \longrightarrow sig_4$ and $sig_3 \sqsubseteq sig_4$.

Monotonic Abstraction

Definition

A transition system T is monotonic if the following holds. For any states sig_1, sig_2 and sig_3 such that $sig_1 \sqsubseteq sig_2$ and $sig_1 \longrightarrow sig_3$, we can always find a state sig_4 such that $sig_2 \longrightarrow sig_4$ and $sig_3 \sqsubseteq sig_4$.

- Our Transition System is not monotonic

Monotonic Abstraction

Definition

A transition system T is monotonic if the following holds. For any states sig_1, sig_2 and sig_3 such that $sig_1 \sqsubseteq sig_2$ and $sig_1 \longrightarrow sig_3$, we can always find a state sig_4 such that $sig_2 \longrightarrow sig_4$ and $sig_3 \sqsubseteq sig_4$.

- Our Transition System is not monotonic
- Construct an over approximation

Monotonic Abstraction

Definition

A transition system T is monotonic if the following holds. For any states sig_1, sig_2 and sig_3 such that $sig_1 \sqsubseteq sig_2$ and $sig_1 \longrightarrow sig_3$, we can always find a state sig_4 such that $sig_2 \longrightarrow sig_4$ and $sig_3 \sqsubseteq sig_4$.

- Our Transition System is not monotonic
- Construct an over approximation
- If the over approximation is valid, the original will also be valid

Monotonic Abstraction

Definition

A transition system T is monotonic if the following holds. For any states sig_1, sig_2 and sig_3 such that $sig_1 \sqsubseteq sig_2$ and $sig_1 \longrightarrow sig_3$, we can always find a state sig_4 such that $sig_2 \longrightarrow sig_4$ and $sig_3 \sqsubseteq sig_4$.

- Our Transition System is not monotonic
- Construct an over approximation
- If the over approximation is valid, the original will also be valid

Definition

We say $s \longrightarrow_A s'$ for two signatures s and s' iff there is an s" such that s" $\sqsubseteq s$ and $s \longrightarrow s'$.

Calculating Predecessors

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied
- Definition needed for each operation *op*:

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied
- Definition needed for each operation *op*:
  - $pre(\mathrm{x} = \mathrm{malloc()})(sig)$

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied
- Definition needed for each operation *op*:
  - $pre(\mathrm{x} = \mathrm{malloc}())(sig)$
  - $pre(\mathrm{free}(\mathrm{x}))(sig)$

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to sig if op is applied
- Definition needed for each operation op:
  - $pre(\mathrm{x} = \mathrm{malloc}())(sig)$
  - $pre(\mathrm{free}(\mathrm{x}))(sig)$
  - $pre(\mathrm{x} == \mathrm{y})(sig)$

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to sig if op is applied
- Definition needed for each operation op:
  - $pre(\mathrm{x} = \mathrm{malloc}())(sig)$
  - $pre(\mathrm{free}(\mathrm{x}))(sig)$
  - $pre(\mathrm{x} == \mathrm{y})(sig)$
  - $pre(\mathrm{x} \mathrel{!}= \mathrm{y})(sig)$

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied
- Definition needed for each operation *op*:
  - $pre(\mathrm{x} = \mathrm{malloc}())(sig)$
  - $pre(\mathrm{free(x)})(sig)$
  - $pre(\mathrm{x} == \mathrm{y})(sig)$
  - $pre(\mathrm{x} \mathrel{!}= \mathrm{y})(sig)$
  - $pre(\mathrm{x} = \mathrm{y.next(i)})(sig)$

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied
- Definition needed for each operation *op*:
  - $pre(\mathrm{x} = \mathrm{malloc()})(sig)$
  - $pre(\mathrm{free(x)})(sig)$
  - $pre(\mathrm{x} == \mathrm{y})(sig)$
  - $pre(\mathrm{x} \mathrel{!}= \mathrm{y})(sig)$
  - $pre(\mathrm{x} = \mathrm{y.next(i)})(sig)$
  - $pre(\mathrm{x.next(i)} = \mathrm{y})(sig)$

Calculating Predecessors

The predecessor function $pre(op)(sig)$:

- Calculates the set of signatures which leads to *sig* if *op* is applied
- Definition needed for each operation *op*:
  - $pre(\mathrm{x} = \mathrm{malloc}())(sig)$
  - $pre(\mathrm{free}(\mathrm{x}))(sig)$
  - $pre(\mathrm{x} == \mathrm{y})(sig)$
  - $pre(\mathrm{x}\ != \mathrm{y})(sig)$
  - $pre(\mathrm{x} = \mathrm{y.next(i)})(sig)$
  - $pre(\mathrm{x.next(i)} = \mathrm{y})(sig)$
  - $pre(\mathrm{x} = \mathrm{y})(sig)$

Reachability Algorithm

Reachability Algorithm

- Backwards Analysis from set of bad configurations S_{bad}

Reachability Algorithm

- Backwards Analysis from set of bad configurations S_{bad}
- Compute predecessors for every $s \in S_{bad}$

Reachability Algorithm

- Backwards Analysis from set of bad configurations S_{bad}
- Compute predecessors for every $s \in S_{bad}$
- If a predecessor p is computed with $s' \sqsubseteq p$, where s' is a previously calculated signature, discard p

Reachability Algorithm

- Backwards Analysis from set of bad configurations S_{bad}
- Compute predecessors for every $s \in S_{bad}$
- If a predecessor p is computed with $s' \sqsubseteq p$, where s' is a previously calculated signature, discard p
- Repeat until all new computed predecessors are being discarded

Reachability Algorithm

- Backwards Analysis from set of bad configurations S_{bad}
- Compute predecessors for every $s \in S_{bad}$
- If a predecessor p is computed with $s' \sqsubseteq p$, where s' is a previously calculated signature, discard p
- Repeat until all new computed predecessors are being discarded
- If the set of signatures is disjoint from the initial set of states $\Rightarrow$ program execution is safe

Conclusion

Conclusion

- Approach has been implemented in a Java prototype

Conclusion

- Approach has been implemented in a Java prototype

Table 1. Experimental results

Program	Struct	Time	#Sig.
Traverse	DLL	11.4 s	294
Insert	DLL	3.5 s	121
Ordered Insert	DLL	19.4 s	793
Merge	DLL	6 min 40 s	8171
Reverse	DLL	10.8 s	395
Search	Tree	1.2 s	51
Insert	Tree	6.8 s	241

Prototype Results [ACV13]

Conclusion

Conclusion

- Relatively simple method

Conclusion

- Relatively simple method
- Very generic approach

Conclusion

- Relatively simple method
- Very generic approach
- Successfully implemented

Conclusion

Any questions?

Parosh Aziz Abdulla, Jonathan Cederberg, and Tomáš Vojnar, *Monotonic abstraction for programs with multiply-linked structures*, International Journal of Foundations of Computer Science **24** (2013), no. 02, 187–210.