Analysing Heap Manipulating Programs: An Automata-theoretic Approach

Supratik Chakraborty
IIT Bombay

November 13, 2012

Outline of Talk

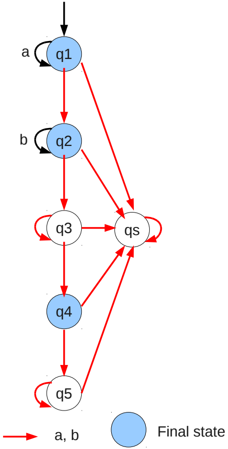
- Some automata basics
- Programs, heaps and analysis
- Regular model checking

Some Automata Basics

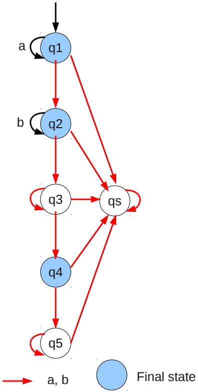
Finite State Automata

A 5−tuple $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, where

- Q : Finite set of states
- Σ : Input alphabet
- $Q_0 \subseteq Q$: Initial states
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$: State transition relation
- F : Set of final states

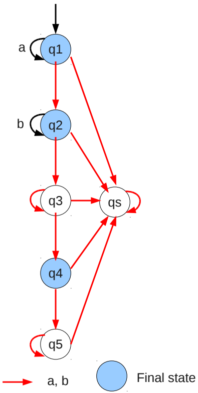


Runs and acceptance



Runs and acceptance

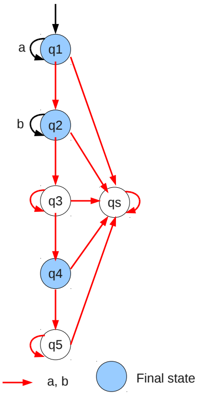
- An finite word $\alpha \in \Sigma^*$



- $\alpha = abbbbb$

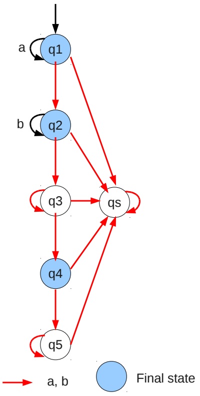
Runs and acceptance

- An finite word $\alpha \in \Sigma^*$
- A *run* of $\mathcal{A}$ on α is a sequence $\rho : \mathbb{N} \to Q$ such that
  - $\rho(0) \in Q_0$
  - $\rho(i+1) \in \delta(\rho(i), \alpha(i))$



- $\alpha = abbbbb$
- $\rho_1 = q_1 q_2 q_2 q_2 q_2 q_2 q_2$

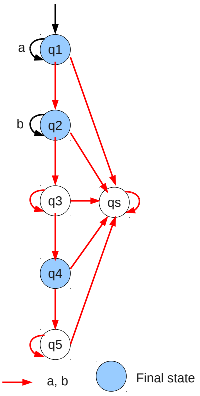
- An finite word $\alpha \in \Sigma^*$
- A *run* of $\mathcal{A}$ on α is a sequence $\rho : \mathbb{N} \rightarrow Q$ such that
  - $\rho(0) \in Q_0$
  - $\rho(i+1) \in \delta(\rho(i), \alpha(i))$
- An automaton may have many runs on α.



- $\alpha = abbbbb$
- $\rho_1 = q_1 q_2 q_2 q_2 q_2 q_2 q_2$
- $\rho_2 = q_1 q_1 q_2 q_2 q_2 q_2 q_2$

Runs and acceptance

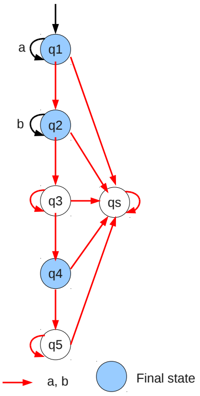
- An finite word $\alpha \in \Sigma^*$
- A *run* of $\mathcal{A}$ on α is a sequence $\rho : \mathbb{N} \to Q$ such that
  - $\rho(0) \in Q_0$
  - $\rho(i+1) \in \delta(\rho(i), \alpha(i))$
- An automaton may have many runs on α.
- ρ is *accepting* iff $\rho(|\alpha|) \in F$



- $\alpha = abbbbb$
- $\rho_1 = q_1 q_2 q_2 q_2 q_2 q_2 q_2$
- $\rho_2 = q_1 q_1 q_2 q_2 q_2 q_2 q_2$

Runs and acceptance

- An finite word $\alpha \in \Sigma^*$
- A *run* of $\mathcal{A}$ on α is a sequence $\rho : \mathbb{N} \to Q$ such that
  - $\rho(0) \in Q_0$
  - $\rho(i+1) \in \delta(\rho(i), \alpha(i))$
- An automaton may have many runs on α.
- ρ is *accepting* iff $\rho(|\alpha|) \in F$
- α is accepted by $\mathcal{A}$ ($\alpha \in L(\mathcal{A})$) iff there is at least one accepting run of $\mathcal{A}$ on α.

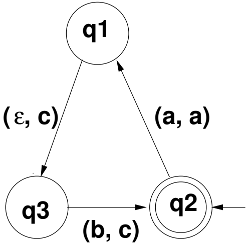


- $\alpha = abbbbb$
- $\rho_1 = q_1 q_2 q_2 q_2 q_2 q_2 q_2$
- $\rho_2 = q_1 q_1 q_2 q_2 q_2 q_2 q_2$

Finite State Transducer (FST)

A 6-tuple
$\tau = (Q, \Sigma_1, \Sigma_2, Q_0, \delta_\tau, F)$

- Q: Set of states
- Σ_1: Input alphabet
- Σ_2: Output alphabet
- $Q_0 \subseteq Q$: Initial set of states
- $\delta_\tau \subseteq Q \times (\Sigma_1 \cup \{\varepsilon\}) \times (Sigma_2 \cup \{\varepsilon\}) \times Q$: Transition relation
- F: Set of final states



- Transduces *ab* to *acc*
- Goes from q_1 to q_2 on input *ab* and outputs *acc*

Regular Relations

$\tau = (Q, \Sigma_1, \Sigma_2, Q_0, \delta_\tau, F)$: Finite state transducer

- Binary relation R_τ:
  - $\{(u, v) \mid u \in \Sigma_1^*, v \in \Sigma_2^*, \exists q \in Q_0, \exists q' \in F,$ q' can be reached from q on reading u and producing $v\}$
- Image under R_τ:
  - Given $L \subseteq \Sigma_1^*$, define $R_\tau(L) = \{v \mid \exists u \in L, (u, v) \in R_\tau\}$
- Composition:
  - $R_1 \circ R_2 = \{(u, v) \mid \exists x,\ (u, x) \in R_1 \text{ and } (x, v) \in R_2\}$
  - Requires output alphabet of R_1 same as input alphabet of R_2.
  - Can compose R_τ with itself if $\Sigma_1 = \Sigma_2$
- Iterated composition: R_τ with $\Sigma_1 = \Sigma_2 = \Sigma$
  - $id = \{(u, u) \mid u \in \Sigma^*\}$: identity relation
  - $R_\tau^0 = id$
  - $R_\tau^{i+1} = R_\tau \circ R_\tau^i$, for all $i \geq 0$
  - $R_\tau^* = \bigcup_{i \geq 0} R_\tau^i$

Programs, Heaps and Analysis

Programs and Heaps

What is a “heap”?

- Informally: Logical pool of memory locations
- Formally: A *partial* map of MemoryLocations to Values

A heap-manipulating program:

```
     func(hd, x)
     // all vars of ptr type
L1:  t1 := hd;
L2:  while (not(t1 = nil)) do
L3:    if (t1 = x) then
L4:        t2 := new;
L5:        t3 := x->n;
L6:        t2->n := t3;
L7:        x->n := t2;
L8:        t1 := t1->n;
L9:    else t1 := t1-> n;
L10: return;
```

A concrete problem

Given a sequential program that manipulates dynamic linked data structures by creating/deleting memory cells and by updating links between them, how do we prove assertions about the resulting structures in heap (trees, lists, ...)?

- Undecidable in general
  - Represent non-blank part of TM tape as doubly-linked list
  - Ask if the tape ever becomes completely blank

Reasoning about Heaps

A concrete problem

Given a sequential program that manipulates dynamic linked data structures by creating/deleting memory cells and by updating links between them, how do we prove assertions about the resulting structures in heap (trees, lists, ...)?

- Undecidable in general
  - Represent non-blank part of TM tape as doubly-linked list
  - Ask if the tape ever becomes completely blank
- But that doesn't reduce the importance of the problem

Reasoning about Heaps

A concrete problem

Given a sequential program that manipulates dynamic linked data structures by creating/deleting memory cells and by updating links between them, how do we prove assertions about the resulting structures in heap (trees, lists, ...)?

- Undecidable in general
  - Represent non-blank part of TM tape as doubly-linked list
  - Ask if the tape ever becomes completely blank
- But that doesn't reduce the importance of the problem
- Can we solve special cases of the problem?

Reasoning about Heaps

A concrete problem

Given a sequential program that manipulates dynamic linked data structures by creating/deleting memory cells and by updating links between them, how do we prove assertions about the resulting structures in heap (trees, lists, ...)?

- Undecidable in general
  - Represent non-blank part of TM tape as doubly-linked list
  - Ask if the tape ever becomes completely blank
- But that doesn't reduce the importance of the problem
- Can we solve special cases of the problem?
- **YES!** for some important special cases
  - Several techniques in literature
  - This talk only about some automata-theoretic techniques
  - Other powerful techniques exist (including automata-based)

Some Simplifying Assumptions

- Heap allocated objects have selectors, e.g.`x->n`
  - Assume one selector per object

Some Simplifying Assumptions

- Heap allocated objects have selectors, e.g.`x->n`
  - Assume one selector per object
- Focus on link structures, abstract (ignore) other data types
  - Sole abstract data type: pointer to memory location
  - Simple `new` and `free` sufficient

Some Simplifying Assumptions

- Heap allocated objects have selectors, e.g.`x->n`
  - Assume one selector per object
- Focus on link structures, abstract (ignore) other data types
  - Sole abstract data type: pointer to memory location
  - Simple `new` and `free` sufficient
- No long sequences of selectors
  - `x->n->n := y->n->n;` semantically equivalent to
  - `temp1 := x->n; temp2 := y->n; temp3 := temp2->n; temp1->n := temp3;`
  - `temp1`, `temp2`, `temp3` fresh variables.

Some Simplifying Assumptions

- Heap allocated objects have selectors, e.g.`x->n`
  - Assume one selector per object
- Focus on link structures, abstract (ignore) other data types
  - Sole abstract data type: pointer to memory location
  - Simple `new` and `free` sufficient
- No long sequences of selectors
  - `x->n->n := y->n->n;` semantically equivalent to
  - `temp1 := x->n; temp2 := y->n; temp3 := temp2->n; temp1->n := temp3;`
  - `temp1`, `temp2`, `temp3` fresh variables.
- Simplify garbage handling
  - Garbage: Allocated memory in heap, no means of access
  - Example: `x := new; x:= new;`
  - Treat garbage generation as error/assume *garbage collection*
  - Rest of analysis assumes no garbage

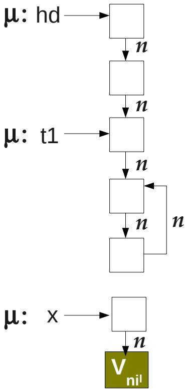
A Simple Imperative Language

PVar	::=	u | v | . . . (pointer-valued variables)
FName	::=	n | f | . . . (pointer-valued selectors)
PExp	::=	PVar | PVar->FName
BExp	::=	PVar = PVar | Pvar = **nil** | **not** BExp |
		BExp **or** BExp | BExp **and** BExp
Stmt	::=	AsgnStmt | CondStmt | LoopStmt |
		SeqCompStmt | AllocStmt | FreeStmt
AsgnStmt	::=	PExp := PVar | PVar := PExp | PExp := **nil**
AllocStmt	::=	PVar := **new**
FreeStmt	::=	**free**(PVar)
CondStmt	::=	**if** (BoolExp) **then** Stmt **else** Stmt
LoopStmt	::=	**while** (BoolExp) **do** Stmt
SeqCompStmt	::=	Stmt ; Stmt

Heap Graph

Given program P with variable names in Σ_P and selector names in Σ_f, construct $G = (V, E, v_{nil}, \lambda, \mu)$

- V: Memory locations allocated by P
- v_{nil}: Represents "nil" value
- $E \subseteq \; V \setminus \{v_{nil}\} \times V$: Link structure
- $\lambda : E \rightarrow 2^{\Sigma_f} \setminus \{\emptyset\}$: Selector assignments
- $\mu : \Sigma_p \hookrightarrow V$: (Partial) variable assignments



- Program state *(minimalist view)*:
  - Location of statement to execute (pc)
  - Representation of heap graph

Analyzing Programs with Heaps

- Program state *(minimalist view)*:
  - Location of statement to execute (pc)
  - Representation of heap graph
- Why not construct a state transition graph?
  - Finite no. of locations: Good!
  - Unbounded vertices in heap graph: Bad!

Analyzing Programs with Heaps

- Program state *(minimalist view)*:
  - Location of statement to execute (pc)
  - Representation of heap graph
- Why not construct a state transition graph?
  - Finite no. of locations: Good!
  - Unbounded vertices in heap graph: Bad!
- Represent (unbounded) heap graph smartly
- Effectively reason about the representation

Regular Model Checking

Regular (Word) Model Checking (RMC)

- Represent heap graph (more generally, state) as finite (unbounded) words on a finite alphabet Σ
  - Brass tacks coming soon!
- Set of states $\subseteq \Sigma^*$
  - A language!
  - If regular, use a finite-state automaton
- Executing a program statement transforms one state (word) to another (word)
  - State transition relation is a word transducer
  - Is it a finite-state transducer?

Regular (Word) Model Checking (RMC)

- Represent heap graph (more generally, state) as finite (unbounded) words on a finite alphabet Σ
  - Brass tacks coming soon!
- Set of states $\subseteq \Sigma^*$
  - A language!
  - If regular, use a finite-state automaton
- Executing a program statement transforms one state (word) to another (word)
  - State transition relation is a word transducer
  - Is it a finite-state transducer?
  - Yes! for several classes of programs

Core Idea of RMC (with words)

- Program states (not just heap graphs): Finite words
- Operational semantics
  - Program statement: Finite state transducer over words
  - Program: Non-deterministically compose transducers for all statements to give a larger transducer τ
- Regular set of initial and "error" program states: I and Bad
- $R_\tau^*(I) = \bigcup_{i \geq 0} R_\tau^i(I)$ denotes set of all reachable states
  - $R_\tau^*(I)$ may not be regular, even if R_τ and I are regular
  - Common solution: Regular overapproximations
- Check if $R_\tau^*(I) \cap Bad = \emptyset$

Core Idea of RMC (with words)

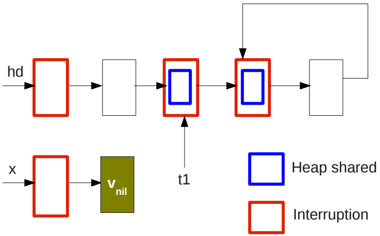
- Program states (not just heap graphs): Finite words
- Operational semantics
  - Program statement: Finite state transducer over words
  - Program: Non-deterministically compose transducers for all statements to give a larger transducer τ
- Regular set of initial and "error" program states: I and Bad
- $R^*_\tau(I) = \bigcup_{i \geq 0} R^i_\tau(I)$ denotes set of all reachable states
  - $R^*_\tau(I)$ may not be regular, even if R_τ and I are regular
  - Common solution: Regular overapproximations
- Check if $R^*_\tau(I) \cap Bad = \emptyset$

Focus of subsequent talk

- Encoding states as finite words
- Operational semantics of program statement
- Overapproximating $R^*_\tau(I)$

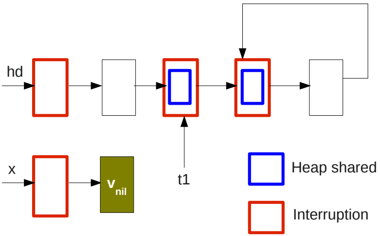
Properties of Heap Graphs

- Recall: Single pointer-valued selector of heap-allocated objects
- Heap graph: Singly linked lists with possible sharing of elements and circularly linked structures



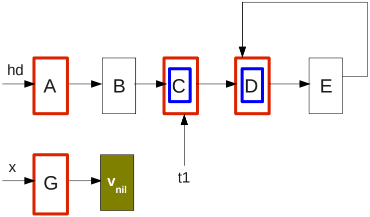
Properties of Heap Graphs

- Recall: Single pointer-valued selector of heap-allocated objects
- Heap graph: Singly linked lists with possible sharing of elements and circularly linked structures



- Heap shared nodes
  - Two (or more) incoming edges, or
  - One incoming edge + pointed to by variable
- Interruption: heap-shared node or pointed to by variable

Properties of Heap Graphs



Observation [Manevich et al' 2005]

With n program variables, heap graph has $\leq n$ heap shared nodes, $\leq 2n$ interruptions, $\leq 2n$ uninterrupted lists

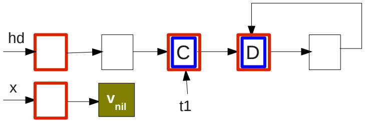
Example: $A \rightarrow B \rightarrow C$, $C \rightarrow D$, $D \rightarrow E \rightarrow D$, $G \rightarrow V_{nil}$

Encoding Heap Graphs as Words

Heap graph: Set of uninterrupted lists

Encoding

- Assign unique name from rank-ordered set to each heap-shared node
- Uninterrupted list from heap-shared node C with 1 link (sequence of n selectors) to heap-shared node D: $C.nD$
- Use $\top$ ($\bot$) to denote uninitialized (nil) terminated lists
- List encodings of uninterrupted lists separated by $|$



Ordering of names
$hd \prec t1 \prec x \prec C \prec D$.
Encoding:
$hd.n.nt1C \mid t1C.nD \mid D.n.nD \mid x.n\bot$

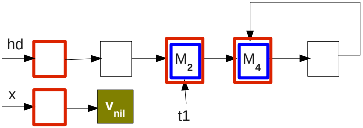
Encoding States

- k program variables
- $\Sigma_M = \{M_0, M_1, M_2, \ldots M_k\}$: rank-ordered names for heap-shared nodes
- Σ_p: Set of program variable names
- Σ_L: Set of program locations (pc values)
- $\Sigma_C = \{C_N, C_0, C_1, C_2, \ldots C_k\}$: mode flags

Encoding States

- k program variables
- $\Sigma_M = \{M_0, M_1, M_2, \ldots M_k\}$: rank-ordered names for heap-shared nodes
- Σ_p: Set of program variable names
- Σ_L: Set of program locations (pc values)
- $\Sigma_C = \{C_N, C_0, C_1, C_2, \ldots C_k\}$: mode flags
- Program state: $w = |w_1|w_2|w_3|w_4|w_5|$, where
  - $|$ doesn't appear in any w_1, w_2, w_3, w_4
  - w_5 encodes heap-graph: word over $\Sigma_M \cup \{\top, \bot, |, .n\}$
  - $w_1 \in \Sigma_C \cdot \Sigma_L$: mode + program location
  - w_2: (Possibly empty) rank-ordered sequence of unused names for heap-shared nodes
  - w_3: (Possibly empty) rank-ordered sequence of uninitialized variable names
  - w_4: (Possibly empty) rank-ordered sequence of variable names set to *nil*
  - w: Finite word over $\Sigma_C \cup \Sigma_L \cup \Sigma_M \cup \Sigma_p \cup \{\top, \bot, |, .n\}$

Encoding states



- Consider earlier program at L9 and above heap graph with variables t2, t3 uninitialized
  - 5 program variables, so $\Sigma_M = \{M_0, M_1, M_2, M_3, M_4, M_5\}$
  - State:
    $| \ C_N \ L9 \ | \ M_0 \ M_3 \ M_4 \ M_5 \ | \ t2 \ t3 \ | \ | \ hd.nM_1 \ | t1 M_1.nM_2 \ | \ xM_2.n.n\bot \ |$

Purpose of Mode Flags

For program with heap-shared node names in $\Sigma_M = \{M_0, M_1, \ldots M_k\}$

- Mode flags in $\Sigma_C = \{C_N, C_0, C_1, \ldots C_k\}$
- C_N : Normal mode of operation
- $C_i, i \in \{0, \ldots k\}$: Mode for reclaiming name M_i
  - Reclaim name of heap-shared node once it ceases to be heap-shared
  - Crucial to be able to work with finite set of names

Operational Semantics of Statements

- Finite state word transducers
- Two special "sink" states: q_{mem} and q_{err}
  - Go to q_{mem} if garbage is generated, *nil* or uninitialized pointer dereferenced
  - Go to q_{err} on realizing that we made a wrong move sometime in the past
- Simple for assignment, allocation and de-allocation statements
- Use non-deterministic guesses to encode semantics of conditional and loop statements
  - Recall state: $|w_1|w_2|w_3|w_4|w_5|$, where w_5 encodes heap
  - Can't determine next location until we've seen whole of w
  - So, how do we figure out values of w_1, w_2, w_3, w_4 in next state?

Operational Semantics of Statements

- Finite state word transducers
- Two special "sink" states: q_{mem} and q_{err}
  - Go to q_{mem} if garbage is generated, *nil* or uninitialized pointer dereferenced
  - Go to q_{err} on realizing that we made a wrong move sometime in the past
- Simple for assignment, allocation and de-allocation statements
- Use non-deterministic guesses to encode semantics of conditional and loop statements
  - Recall state: $|w_1|w_2|w_3|w_4|w_5|$, where w_5 encodes heap
  - Can't determine next location until we've seen whole of w
  - So, how do we figure out values of w_1, w_2, w_3, w_4 in next state?
  - Non-deterministically guess, remember guess in finite control, check as rest of word is read, transition to q_{err} if guess incorrect

Computing (approximate) $R_\tau^*(I)$

- Quotienting techniques
- Abstraction-refinement techniques
- Extrapolation/widening techniques
- Regular language inferencing techniques