Verification of Recursive Methods on Tree-like Data Structures

Jyotirmoy V. Deshmukh E. Allen Emerson
{deshmukh,emerson}@cs.utexas.edu

University of Texas at Austin

Formal Methods in Computer-Aided Design 2009

Recursive Methods are Everywhere!

- Data Structure Libraries.
- File Systems.
- BDD packages.
- Netlist Manipulation Routines.

Recursive Method: changeData

```
void changeData (iter) {
   if ((iter->next₁==∅) && (iter->next₂==∅)) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next₁!=∅) { changeData (iter->next₁); }
   incMod3 (iter->data);
   if (iter->next₂!=∅) { changeData (iter->next₂); }
   incMod3 (iter->data);
   return;
}
```

```
void incMod3 (x) {
   return (x + 1) mod 3;
}
```

Properties of Interest

Sample Pre-Condition

Input is a binary tree, data values in $\{0, 1, 2\}$.

Sample Post-Condition(s)

(A) Output is an acyclic data structure.
(B) Output is a binary tree (subsumes (A)).
(C) Leaf nodes in Output incremented by one (mod 3).
(D) Non-leaf nodes in Output remain unchanged.

Verification instance of the Parameterized Reasoning problem.

General Methods and Properties

In general, methods could . . .

- Change links.
- Add nodes.
- Delete nodes.

For example, specifications could be . . .

- Sorted-ness in a list.
- Left key is less than Right key.
- Both children of every red node are black.
- All leaves are black.

Outline

1. Scope
2. Method Automata
3. Verification Framework
4. Complexity and Results

Outline

Most General Recursive Method over a Tree...

Signature:

- Arbitrary pointer arguments, data arguments.
- Pointer/Data value as return value.

Body: (in no particular order)

- Assignments to pointer expressions.
- Recursive calls.
- Access to global pointer/data values.

Decidable Fragment

An arbitrary recursive method can simulate a Turing Machine.

Syntactic restrictions for decidability?

Disallow:

- Global pointer variables.
  (...else method models k-pebble automaton)
- Pointers arbitrarily far apart.
  (...else method models k-headed automaton)
- Unbounded destructive changes.
  (...else method models linear bounded automaton)

Decidable Fragment

Syntactic restrictions for decidability?

Disallow:

- ~~Global pointer variables.~~

  ~~(. . . else method models k-pebble automata)~~
- Pointers arbitrarily far apart.

  (. . . else method models k-headed automata)
- Unbounded destructive changes.

  (. . . else method models Linear Bounded Automata)

Syntactic Fragment: Updates within a bounded region

Designated pointer argument 'iterator' (**iter**).

Destructive Update relative to **iter**

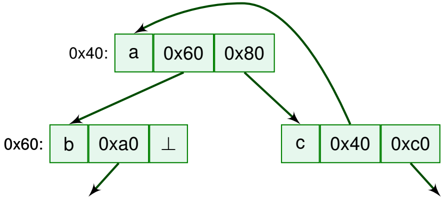
ptr = **iter**, **iter**->**next**$_j$, **iter**->**next**$_j$-> ... ->**next**$_k$.

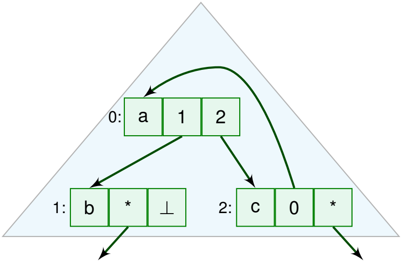
- `ptr->data = d;`
- `ptr->next`$_j$ `= ptr';`
- `ptr->next`$_j$ `= new node(d, ptr`^{1}`, ...ptr`^{k}`);`
- `delete(ptr);`

Windows: Model updates within a bounded distance

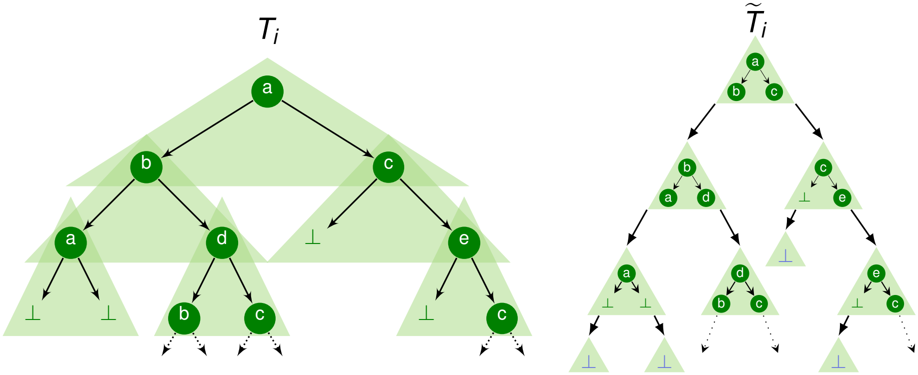
Definition (Window)

- Finite Encoding for *neighborhood* of **node**.
- Concrete address replaced by “Local” address.





Abstract Tree



Obtain T_i from $\widetilde{T}_i$ by eliding everything but the root of each window.

Decidable Fragment

Syntactic restrictions for decidability?

Disallow:

- ~~Global pointer variables.~~

  ~~(. . . else method models k-pebble automata)~~

- ~~Pointers arbitrarily far apart.~~

  ~~(. . . else method models k-headed automata)~~

- Unbounded destructive changes.

  (. . . else method models Linear Bounded Automata)

Syntactic Fragment: Bounded Destructive Updates

Lemma

For trees, ≤ 1 *recursive invocation/child* $\Rightarrow$ *#destructive updates by* $\mathcal{M}$ *bounded.*

Proof.

$\mathcal{M}$ can destructively update n:

(0) when $\mathcal{M}$ first visits n (after invoked from parent of n),

(1) when $\mathcal{M}$ returns from 1^{st} recursive call,

⋮

(K) when $\mathcal{M}$ returns from K^{th} recursive call.

$\Rightarrow$ $\mathcal{M}$ destructively updates n at most $K + 1$ times.
K is fixed for given K-ary tree. □

Decidable Fragment

Syntactic restrictions for decidability?

Disallow:

- ~~Global pointer variables.~~

  ~~(…else method models k-pebble automata)~~
- ~~Pointers arbitrarily far apart.~~

  ~~(…else method models k-headed automata)~~
- ~~Unbounded destructive changes.~~

  ~~(…else method models Linear Bounded Automata)~~

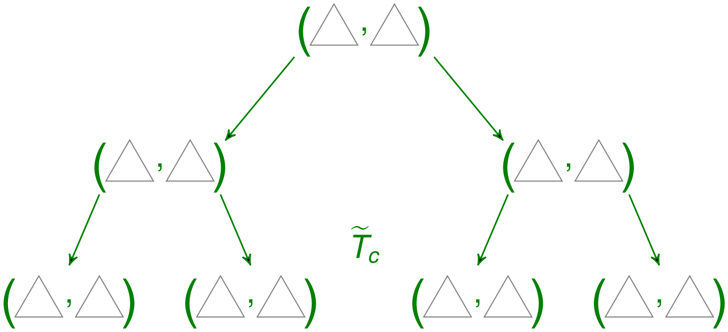
Outline

Template Tail-Recursive Method

```
void foo(iter) {
  if (cond) {
    base-du;
  }
  recur-du;
  foo (iter->next2);
  foo (iter->next1);
  foo (iter->next3);
}
```

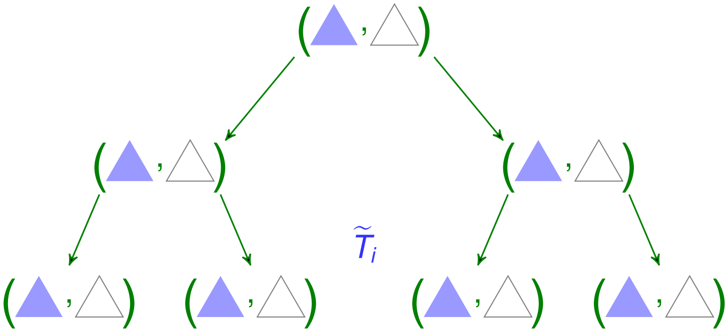
Method Automaton $\mathcal{A}_{\mathcal{M}}$

- $\mathcal{A}_{\mathcal{M}}$ accepts $\widetilde{T}_i \circ \widetilde{T}_o$ iff $T_o = \mathcal{M}(T_i)$.
- $\widetilde{T}_c$ encodes valid actions of $\mathcal{M}$.



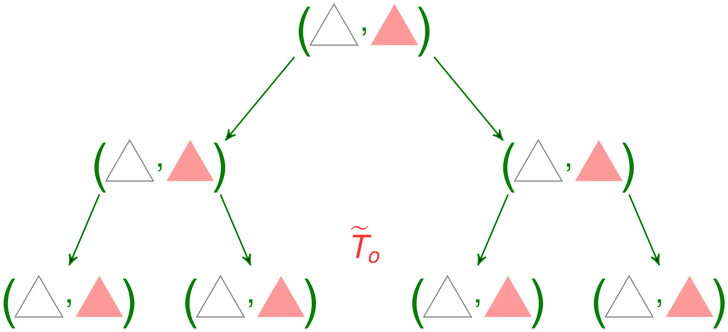
Method Automaton $\mathcal{A}_{\mathcal{M}}$

- $\mathcal{A}_{\mathcal{M}}$ accepts $\widetilde{T}_i \circ \widetilde{T}_o$ iff $T_o = \mathcal{M}(T_i)$.
- $\widetilde{T}_c$ encodes valid actions of $\mathcal{M}$.



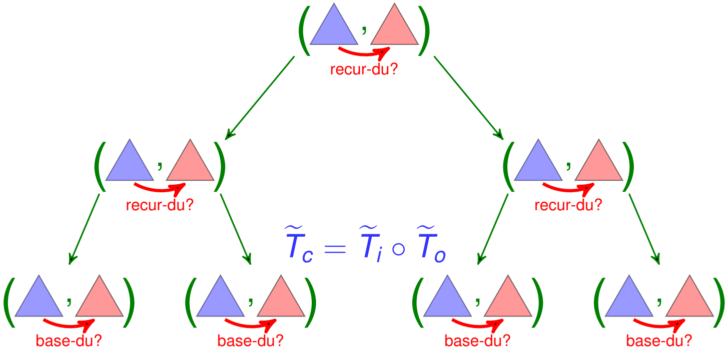
Method Automaton $\mathcal{A}_{\mathcal{M}}$

- $\mathcal{A}_{\mathcal{M}}$ accepts $\widetilde{T}_i \circ \widetilde{T}_o$ iff $T_o = \mathcal{M}(T_i)$.
- $\widetilde{T}_c$ encodes valid actions of $\mathcal{M}$.



Method Automaton $\mathcal{A}_{\mathcal{M}}$

- $\mathcal{A}_{\mathcal{M}}$ accepts $\widetilde{T}_i \circ \widetilde{T}_o$ iff $T_o = \mathcal{M}(T_i)$.
- $\widetilde{T}_c$ encodes valid actions of $\mathcal{M}$.

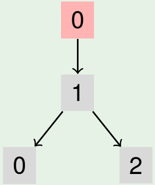


Template Non Tail-Recursive Method

```
void foo(iter) {
  if (cond) {
     base-du;
  }
  recur-du[0];
  foo (iter->next2);
  recur-du[1];
  foo (iter->next1);
  recur-du[2];
  foo (iter->next3);
  recur-du[3];
}
```

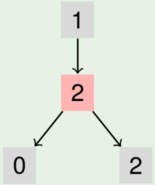
Action of $\mathcal{M}$

```
void changeData (iter) {
   if ((iter->next₁==∅) &&
      (iter->next₂==∅) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next₁!=∅) {
      changeData (iter->next₁);
   }
   incMod3 (iter->data);
   if (iter->next₂!=∅) {
      changeData (iter->next₂);
   }
   incMod3 (iter->data);
   return;
}
```



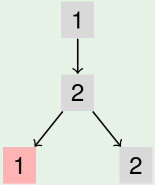
Action of $\mathcal{M}$

```
void changeData (iter) {
   if ((iter->next1==∅) &&
      (iter->next2==∅) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next1!=∅) {
      changeData (iter->next1);
   }
   incMod3 (iter->data);
   if (iter->next2!=∅) {
      changeData (iter->next2);
   }
   incMod3 (iter->data);
   return;
}
```



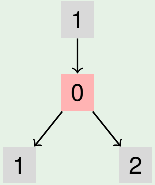
Action of $\mathcal{M}$

```
void changeData (iter) {
   if ((iter->next₁==∅) &&
      (iter->next₂==∅)) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next₁!=∅) {
      changeData (iter->next₁);
   }
   incMod3 (iter->data);
   if (iter->next₂!=∅) {
      changeData (iter->next₂);
   }
   incMod3 (iter->data);
   return;
}
```



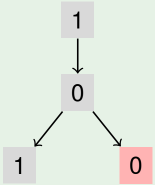
Action of $\mathcal{M}$

```
void changeData (iter) {
    if ((iter->next₁==∅) &&
        (iter->next₂==∅) {
        incMod3(iter->data);
        return;
    }
    incMod3 (iter->data);
    if (iter->next₁!=∅) {
        changeData (iter->next₁);
    }
    incMod3 (iter->data);
    if (iter->next₂!=∅) {
        changeData (iter->next₂);
    }
    incMod3 (iter->data);
    return;
}
```



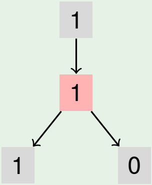
Action of $\mathcal{M}$

```
void changeData (iter) {
   if ((iter->next₁==∅) &&
      (iter->next₂==∅) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next₁!=∅) {
      changeData (iter->next₁);
   }
   incMod3 (iter->data);
   if (iter->next₂!=∅) {
      changeData (iter->next₂);
   }
   incMod3 (iter->data);
   return;
}
```



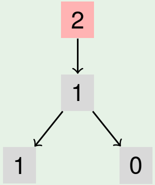
Action of $\mathcal{M}$

```
void changeData (iter) {
   if ((iter->next₁==∅) &&
      (iter->next₂==∅) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next₁!=∅) {
      changeData (iter->next₁);
   }
   incMod3 (iter->data);
   if (iter->next₂!=∅) {
      changeData (iter->next₂);
   }
   incMod3 (iter->data);
   return;
}
```



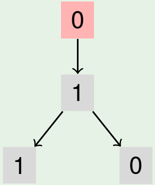
Action of $\mathcal{M}$

```
void changeData (iter) {
   if ((iter->next₁==∅) &&
      (iter->next₂==∅) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next₁!=∅) {
      changeData (iter->next₁);
   }
   incMod3 (iter->data);
   if (iter->next₂!=∅) {
      changeData (iter->next₂);
   }
   incMod3 (iter->data);
   return;
}
```

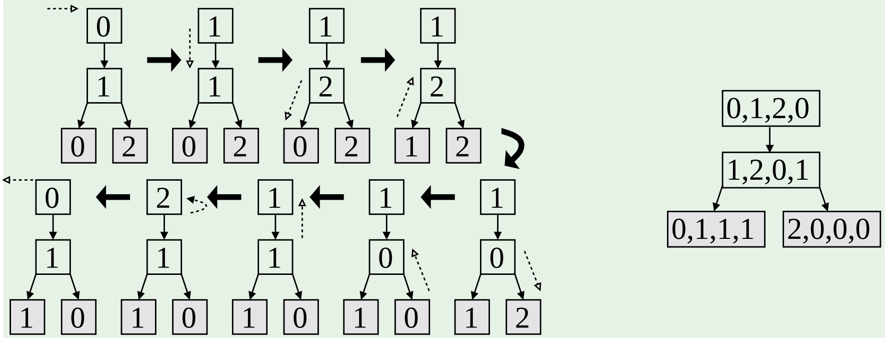


Action of $\mathcal{M}$

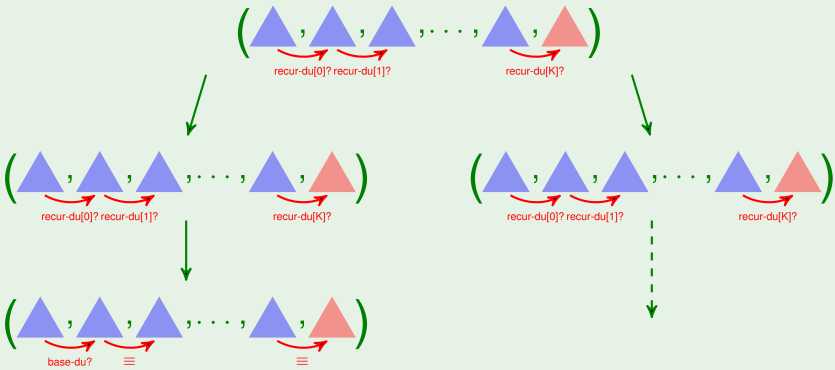
```
void changeData (iter) {
   if ((iter->next1==∅) &&
      (iter->next2==∅)) {
      incMod3(iter->data);
      return;
   }
   incMod3 (iter->data);
   if (iter->next1!=∅) {
      changeData (iter->next1);
   }
   incMod3 (iter->data);
   if (iter->next2!=∅) {
      changeData (iter->next2);
   }
   incMod3 (iter->data);
   return;
}
```



Depth-first action represented by Finite Annotation



Structure of $\mathcal{A}_{\mathcal{M}}$



Outline

1. Scope
2. Method Automata
3. Verification Framework
4. Complexity and Results

Properties as Automata

Pre-condition

- Pre-condition φ provided as $\mathcal{A}_\varphi$.
- $\mathcal{A}_\varphi$ operates on $\widetilde{T}_c = \widetilde{T}_i \circ \widetilde{T}_o$.
- Accepts $\widetilde{T}_i$ if $T_i \models \varphi$.
- Ignores $\widetilde{T}_o$ component of $\widetilde{T}_c$.

Post-condition

- Negated Post-condition ψ provided as $\mathcal{A}_{\neg\psi}$.
- $\mathcal{A}_{\neg\psi}$ operates on $\widetilde{T}_c$.
- Accepts $\widetilde{T}_o$ if $T_o \not\models \psi$.
- Ignores $\widetilde{T}_i$ component of $\widetilde{T}_c$.

Product Automaton

- $\mathcal{A}_p = \mathcal{A}_\mathcal{M} \otimes \mathcal{A}_\varphi \otimes \mathcal{A}_{\neg\psi}$.
- $\mathcal{A}_p$ is *non-empty* $\Leftrightarrow$:
  - $\mathcal{A}_\mathcal{M}$ accepts $\widetilde{T}_i \circ \widetilde{T}_o$, i.e. $T_o = \mathcal{M}(T_i)$,
  - $\mathcal{A}_\varphi$ accepts $\widetilde{T}_i$, i.e. $T_i \models \varphi$,
  - $\mathcal{A}_{\neg\psi}$ accepts $\widetilde{T}_o$, i.e. $T_o \not\models \psi$.
- $\mathcal{A}_p$ is empty $\Leftrightarrow$ $\mathcal{M}$ satisfies pre/post-conditions for all input trees.

Outline

Complexity

- $|\mathcal{A}_p|$ proportional to $|\mathcal{A}_\mathcal{M}|$, properties.
- $|\mathcal{A}_\mathcal{M}|$ linear in $|\mathcal{M}|$, exp. in size of window.
- Overall complexity: polynomial in $|\mathcal{A}_p|$.

Experimental Results

Method	Spec.	Time[a] (secs) $\mathcal{A}_{\mathcal{M}}$	Total	Mem. (MB)
On Linked Lists:				
`DeleteNode`	Acyclic	0.3	1.3	20
`InsertAtTail`	Acyclic	0.01	0.8	<1
`InsertNode`	Acyclic	0.4	1.6	48
On Binary Trees:				
`InsertNode`	Acyclic	15	329	2512
`ReplaceAll(a,b)`	Acyclic	5	26	324
	$\nexists$**`iter`** : **`iter`**$->d = a$	5	27	432
`DeleteLeaf`	Acyclic	12	48	630

[a] Experiments were performed on an Athlon 64X2 4200^{+} system with 6GB RAM.

Thank You!

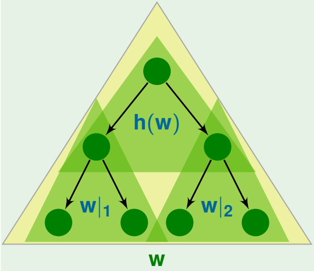
Template for Method in Decidable Fragment

```
void foo (iter, d1, ..., dn) {
  /* base case: */
  if (condition) {
     d.u. to w(iter);
     return;
  }
  /* recursive case: */
  d.u. to w(iter);
  foo (iter->3); // call to 3rd successor
  d.u. to w(iter);
  foo (iter->1); // call to 1st successor
  d.u. to w(iter);
  foo (iter->2); // call to 2nd successor
  d.u. to w(iter);
  return;
}
```

Structure of $\mathcal{A}_\mathcal{M}$: Tail Recursive Methods

- Input symbol $\sigma = (w_i, w_o)$.
- State encodes part of σ overlapping with successor.
- Reads new σ'; rejects if overlapping parts differ.
- If $\sigma \models$ base-case condition, $\mathcal{A}_\mathcal{M}$ accepts if $w_o = \mathcal{M}(w_i)$.
- If $\sigma \not\models$ base-case condition:
  - Checks $w_o \stackrel{?}{=} \mathcal{M}(w_i)$ (rejects otherwise).
  - Transitions to $(w_i|_j, w_o|_j)$ along j^{th} successor.

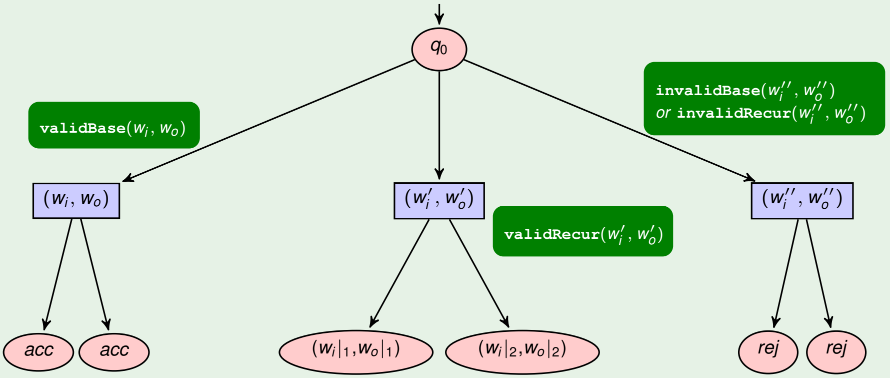
Macros



$$\texttt{consistent}(\underbrace{(x_i, x_o)}_{q}, \underbrace{(w_i, w_o)}_{\sigma}) \stackrel{\text{def}}{=} \underbrace{(h(w_i) = x_i)}_{cons.\ input} \wedge \underbrace{(h(w_o) = x_o)}_{cons.\ output}$$

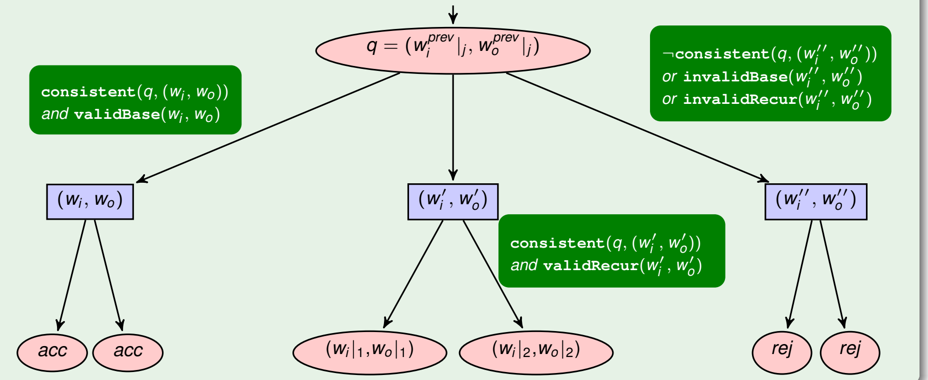
$\mathcal{A}_{\mathcal{M}}$ for Tail-Recursive Methods

Transitions from the Initial State



$\mathcal{A}_\mathcal{M}$ for Tail-Recursive Methods

Transitions from non-initial/final states



More Macros

$$\texttt{validBase}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_i \models \mathbf{bcond})}_{\textit{is base case.}} \wedge \underbrace{(w_o = \mathbf{base\_du}(w_i))}_{\textit{matches } \mathbf{base\_du}}$$

$$\texttt{invalidBase}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_i \models \mathbf{bcond})}_{\textit{is base case.}} \wedge \underbrace{(w_o \neq \mathbf{base\_du}(w_i))}_{\textit{doesn't match } \mathbf{base\_du}}$$

$$\texttt{validRecur}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_i \not\models \mathbf{bcond})}_{\textit{not base case}} \wedge \underbrace{(w_o = \mathbf{recur\_du}(w_i))}_{\textit{matches } \mathbf{recur\_du}}$$

$$\texttt{invalidRecur}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_i \not\models \mathbf{bcond})}_{\textit{not base case}} \wedge \underbrace{(w_o \neq \mathbf{recur\_du}(w_i))}_{\textit{doesn't match } \mathbf{recur\_du}}$$

Non Tail-Recursive Methods

Modified Macros:

$$\texttt{consistent}(\underbrace{(x_i, x_o)}_{q}, \underbrace{(w_0, \ldots, w_{K+1})}_{\sigma}) \stackrel{\text{def}}{=} \underbrace{(h(w_0) = x_i)}_{\textit{cons. input}} \wedge \underbrace{(h(w_{K+1}) = x_o)}_{\textit{cons. output}}$$

$$\texttt{validBase}(w_0, w_1, \ldots, w_{K+1}) \stackrel{\text{def}}{=} \underbrace{(w_0 \models \mathbf{bcond})}_{\textit{is base case.}} \wedge \underbrace{(w_{K+1} = w_K = \ldots = w_1 = \mathbf{base\_du}(w_0))}_{\textit{matches } \mathbf{base\_du}}$$

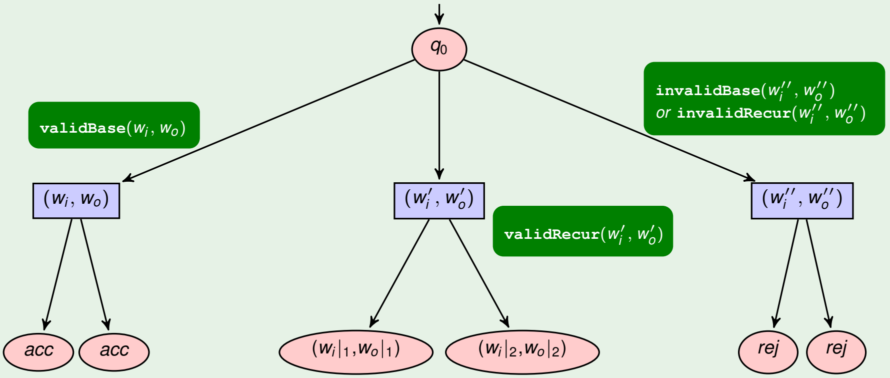
$$\texttt{invalidBase}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_0 \models \mathbf{bcond})}_{\textit{is base case.}} \wedge \underbrace{\neg(w_{K+1} = w_K = \ldots = w_1 = \mathbf{base\_du}(w_0))}_{\textit{doesn't match } \mathbf{base\_du}}$$

$$\texttt{validRecur}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_0 \not\models \mathbf{bcond})}_{\textit{not base case}} \wedge \underbrace{(w_1 = \mathbf{recur\_du[0]}(w_0)) \wedge \ldots \wedge (w_{K+1} = \mathbf{recur\_du[K]}(w_K))}_{\textit{matches all } \mathbf{recur\_du}'s}$$

$$\texttt{invalidRecur}(w_i, w_o) \stackrel{\text{def}}{=} \underbrace{(w_i \not\models \mathbf{bcond})}_{\textit{not base case}} \wedge \underbrace{\neg((w_1 = \mathbf{recur\_du[0]}(w_0)) \wedge \ldots \wedge (w_{K+1} = \mathbf{recur\_du[K]}(w_K)))}_{\textit{doesn't match all } \mathbf{recur\_du}'s}$$

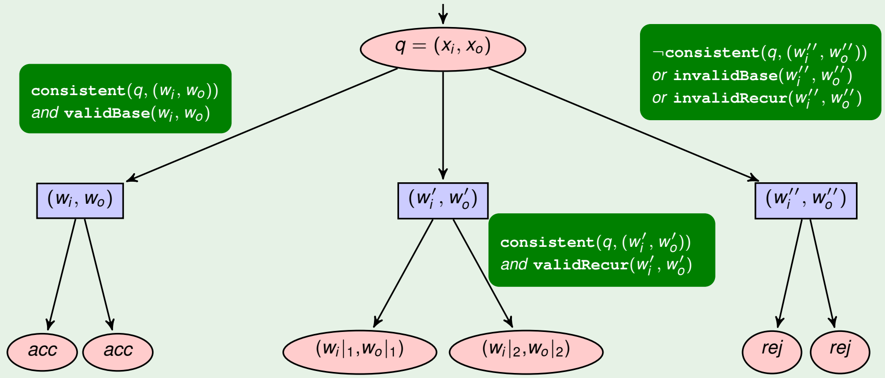
$\mathcal{A}_{\mathcal{M}}$ for Non Tail-Recursive Methods

Transitions from the Initial State



$\mathcal{A}_{\mathcal{M}}$ for Non Tail-Recursive Methods

Transitions from non-initial/final states



Future Work

- Reduce $|\Sigma_p|$ by clustering-based abstraction.
- Verify methods on *dag*s: use single visit property.
- Use of Pushdown/Stack Tree Automata as Method Automata.