The MONA Project

Logic, Automata, and Program Verification

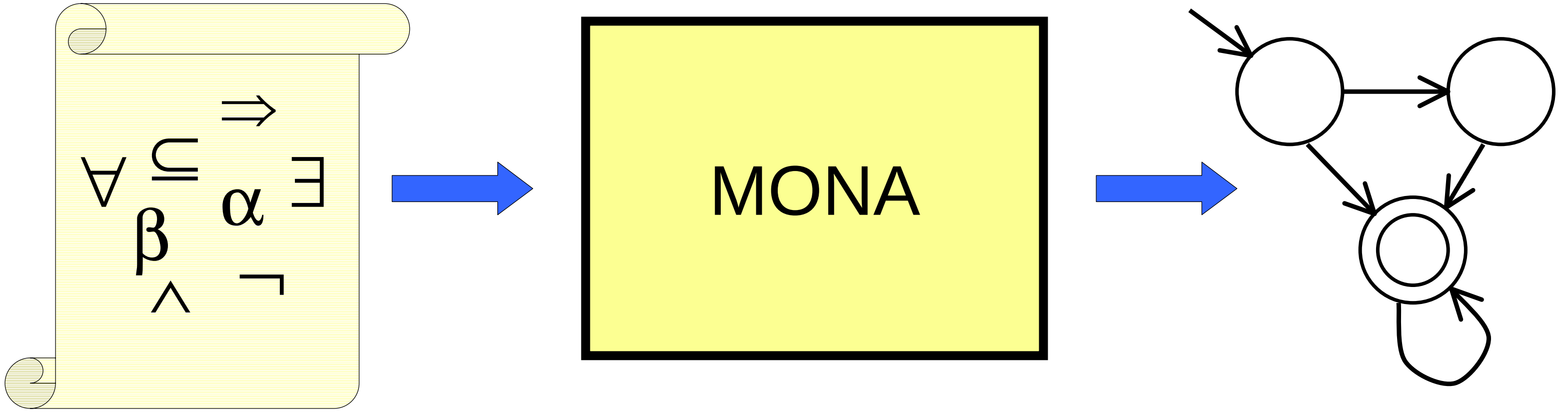
Anders Møller

University of Aarhus

BRICS

http://www.brics.dk/~amoeller/talks/dresden.pdf

The MONA Tool



The MONA tool

- transforms **formulas** into **finite-state automata**
- decides **validity** / provides **counterexamples** for the formulas by analyzing the automata

Example: A Mutual Exclusion Protocol

Hyman's mutual exclusion algorithm:

```
while true do begin
    < noncritical section >
    b_i := true
    while ( k ≠ i ) do begin
        while ( b_{1-i} ) do skip
        k := i
    end
    < critical section >
    b_i := false
end
```

- Two processes executing (i=0 and i=1)
- Hyman's claim: only one can be in the critical section at any time

Example: A Mutual Exclusion Protocol

Encoding the state:

declares variables that range over ***sets of natural numbers***

```
var2 PC0′, PC0′′, PC0′′′, PC1′, PC1′′, PC1′′′, b0, b1, k;

pred p0_at_line_1(var1 t) = t∉PC0′ ∧ t∉PC0′′ ∧ t∉PC0′′′;
pred p0_at_line_2(var1 t) = t∉PC0′ ∧ t∉PC0′′ ∧ t∈PC0′′′;
...

pred b0_false(var1 t) = t∉b0;
pred b0_true(var1 t) = t∈b0;
...

pred k_is_0(var1 t) = t∈k;
pred k_is_1(var1 t) = t∉k;
```

encodes program counters

encodes state

Example: A Mutual Exclusion Protocol

Encoding the dynamics:

```
pred p0_proc_step(var1 t) =
  (p0_at_line_1(t) ⇒ p0_at_line_2(t+1) ∧ unchanged_vars(t)) ∧
  (p0_at_line_2(t) ⇒ p0_at_line_3(t+1) ∧ b0_true(t+1) ∧
                     unchanged_k(t) ∧ unchanged_b1(t)) ∧
  (p0_at_line_3(t) ⇒ (unchanged_vars(t) ∧
                          (k_is_0(t) ⇒ p0_at_line_6(t+1)) ∧
                          (k_is_1(t) ⇒ p0_at_line_4(t+1)))) ∧
  ...
  (p0_at_line_7(t) ⇒ p0_at_line_1(t+1) ∧ b0_false(t+1) ∧
                     unchanged_k(t) ∧ unchanged_b1(t));
...
pred Valid() = p0_at_line_1(0) ∧ p1_at_line_1(0) ∧
  b0_false(0) ∧ b1_false(0) ∧ k_is_1(0) ∧
  (∀1 t: ((p0_proc_step(t) ∧ unchanged_PC1(t))
         | (p1_proc_step(t) ∧ unchanged_PC0(t))));
```

behavior of proc. 0

valid computations

Example: A Mutual Exclusion Protocol

Checking mutual exclusion:

```
Valid() ⇒ ∀¹ t: ¬(p0_at_line_6(t) ∧
                  p1_at_line_6(t));
```

Example: A Mutual Exclusion Protocol

After 0.5 seconds, MONA returns an automaton with 137 states

Reply from MONA automaton analysis:

```
A counterexample of least length (10) is:
PC0'          0  0  0  0  0  1  1  1  0  1
PC0''         0  0  0  1  1  0  0  0  1  0
PC0'''        0  0  1  0  1  0  0  0  0  1
PC1'          0  0  0  0  0  0  0  1  1  1
PC1''         0  0  0  0  0  0  1  0  0  0
PC1'''        0  1  1  1  1  1  0  1  1  1
b0            0  0  0  1  1  1  1  1  1  1
b1            0  0  0  0  0  0  1  1  1  1
k             0  0  0  0  0  0  0  0  1  1
```

This **counterexample** shows the encoding of a valid run which violates the mutual-exclusion property!

Overview

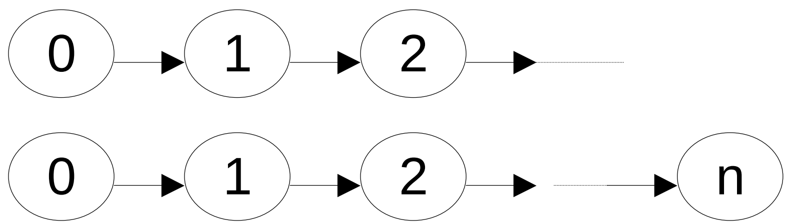
- Introduction: verifying Hyman's mutual exclusion algorithm
- **Mona**dic 2nd-order Logic on finite Strings (M2L-Str) /
  Weak monadic Second-order theory of 1 Successor (WS1S)
- Logic $\rightarrow$ Automata
- Complexity
- Tree logics (M2L-Tree / WS2S)
- Implementation issues
- Applications
- Example: *Pointer Assertion Logic*
- Conclusion

Monadic 2nd-order Logic on Strings

$$
\begin{array}{lll}
\Phi ::= & \neg\Phi \mid \Phi\vee\Phi \mid \Phi\wedge\Phi \mid \Phi\Rightarrow\Phi \mid \Phi\Leftrightarrow\Phi & \\
& \mid \forall^1 x.\Phi \mid \exists^1 x.\Phi \mid \forall^2 X.\Phi \mid \exists^2 X.\Phi & \textit{(formulas)} \\
& \mid t=t \mid t\in T \mid T=T \mid T\subseteq T \mid \ldots & \\
& & \\
T ::= & X \mid T\cup T \mid T\cap T \mid T\setminus T \mid \emptyset & \textit{(set terms)} \\
& & \\
t ::= & x \mid 0 \mid t+1 & \textit{(position terms)}
\end{array}
$$

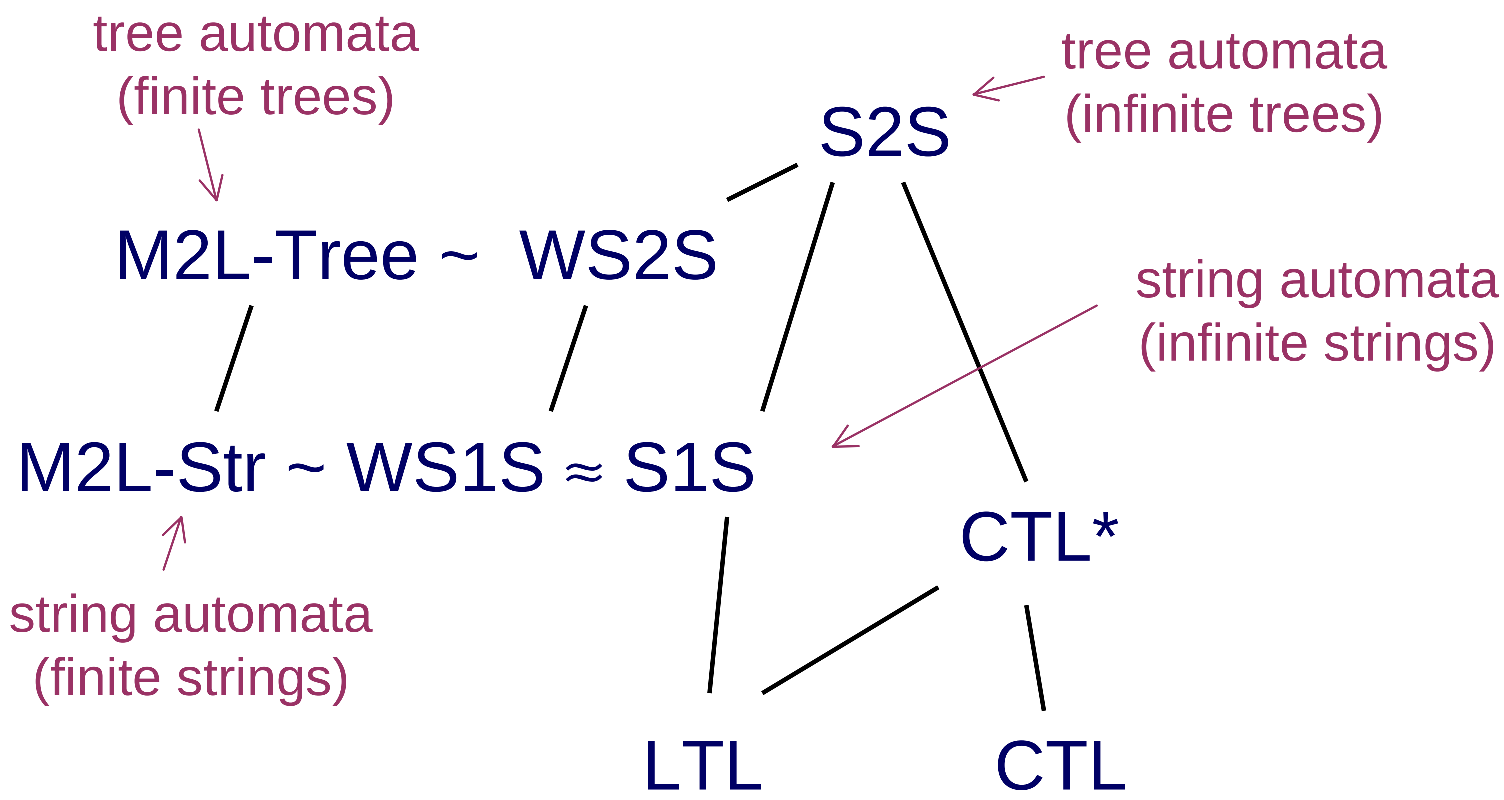
- *Weak Monadic* 2nd-order logic = quantification over *finite sets*

- Two choices for interpretation:
  WS1S: the natural numbers
  M2L-Str: positions in a finite string



- Typical use: as linear temporal logic

Related Logics



Logic $\to$ Automata

Assignment A of values to FV(Φ)

$\Updownarrow$

String w_A over the alphabet $\Sigma=\{0,1\}^k$ where $k=|FV(\Phi)|$

Example:

the assignment $A=[P \mapsto \{2,3\}, Q \mapsto \emptyset, R \mapsto 0]$

corresponds to the string:

$$w_A = \begin{array}{ccccc} \begin{pmatrix}0\\0\\1\end{pmatrix} & \begin{pmatrix}0\\0\\0\end{pmatrix} & \begin{pmatrix}1\\0\\0\end{pmatrix} & \begin{pmatrix}1\\0\\0\end{pmatrix} & \begin{matrix}P\\Q\\R\end{matrix} \\ 0 & 1 & 2 & 3 & \end{array}$$

Define the *language* of Φ : $L(\Phi) = \{ w_A \mid A \models \Phi \}$

Logic $\rightarrow$ Automata

Büchi 1960 / Elgot 1961

Simplified syntax:

Φ ::= $\neg\Phi$ | $\Phi \wedge \Phi$ | $\exists^2 X.\Phi$
| $X_1 \subseteq X_2$ | $X_1 = X_2 \backslash X_3$ | $X_1 = X_2 + 1$

Translation of Φ into automaton A_Φ such that $L(\Phi)=L(A_\Phi)$:

formula Φ	**automaton** A_Φ
atomic formulas	basic automata
negation $\neg$	complement $\complement$
conjunction $\wedge$	intersection $\cap$
existential quantification $\exists$	projection+determinization

– we work with *deterministic minimal* automata

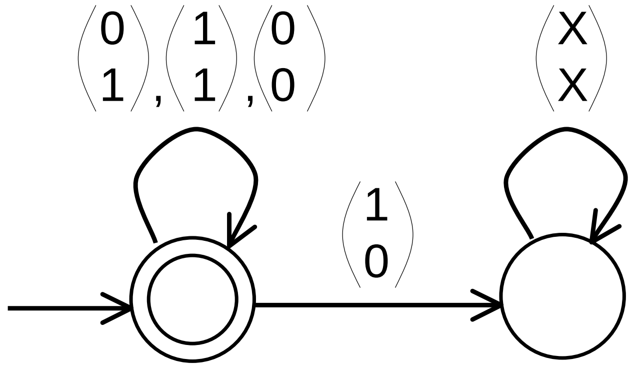
Logic $\rightarrow$ Automata

Example 1:

The atomic formula:

$$\Phi = P \subseteq Q$$

corresponds to the basic automaton:



where P corresponds to the first component, and Q to the second

Logic → Automata

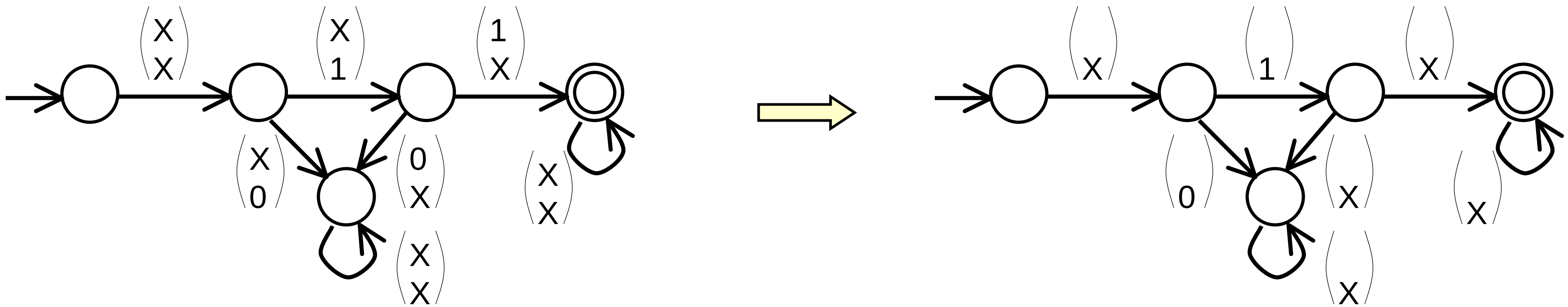
Example 2:

The composite formula:

$$\Phi = \exists^2 P.\Psi$$

corresponds to a **projection** where the P track is removed

Consider $\Phi = \exists^2 P.\ 2 \in P \wedge 1 \in Q$



This is for M2L-Str,

WS1S also needs a **quotient operation** after projection

Automaton Analysis

1. Given a formula Φ, construct the corresponding minimal finite-state automaton A_Φ

2. Look at A_Φ:
   - If $L(A_\Phi)=\Sigma^*$, then Φ is **valid**
   - Otherwise, generate a (minimal) **counter-example** by finding a (minimal) path in A_Φ from the initial state to a non-accepting state

Logic $\leftarrow$ Automata

- Every automaton can be encoded as an M2L-Str formula

– but this direction is not relevant for MONA

Complexity

Practical problems:

- The alphabet size is **exponential** in the number of free variables: $\Sigma=\{0,1\}^{k}$

- A single determinization can cause an exponential increase in state-space size

Worst case: $\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}$ #alternating quantifiers

And it is inevitable: The *decision problem* for WS1S has a **non-elementary** lower bound

Meyer 1972

Only a madman would implement that!

Nils Klarlund

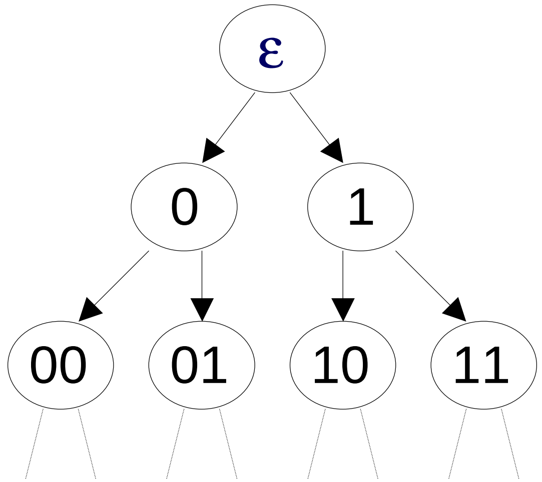
Monadic 2nd-order Logic on Trees

Generalize the structural primitives:

$t \ ::= \ x \mid \cancel{0 \mid t+1} \mid \varepsilon \mid succ_0(t) \mid succ_1(t)$

Interpretation: now over *tree* structures

Thatcher/Wright 1968



Again, two choices of models:

- WS2S: the infinite binary tree
- M2L-Tree: a finite binary tree

Tree Automata

WS2S / M2L-Tree are also decidable using finite-state automata:

A *bottom-up tree automaton* has a transition function of the form

$$\delta\colon Q \times Q \to \Sigma \to Q$$

and assigns a state to each tree node starting from the leaves

- All standard automaton operations (product, minimization, subset construction, ...) generalize elegantly to tree automata
- Extra complexity: a **quadratic** blow-up in the transition function

(Later: an example application encoding
tree-shaped data structures in tree logic...)

Guided Tree Automata (GTA)

– making tree automata practically useful

A GTA ***factorizes*** the state space:

- A user-defined ***guide*** assigns a **state space** to each position in the infinite binary tree

- Each state space has its own transition function

$$\delta_a: Q_b \times Q_c \to \Sigma \to Q_a$$

- This can give an indispensable **exponential** improvement (but it requires a good guide to be defined)

Binary Decision Diagrams (BDDs)

– working with automata with huge alphabets

How to represent transition functions

$$\delta\colon Q \to \Sigma \to Q$$

when $\Sigma = \{0,1\}^k$ and k is large?

The MONA solution:
use **Binary Decision Diagrams** (BDDs)

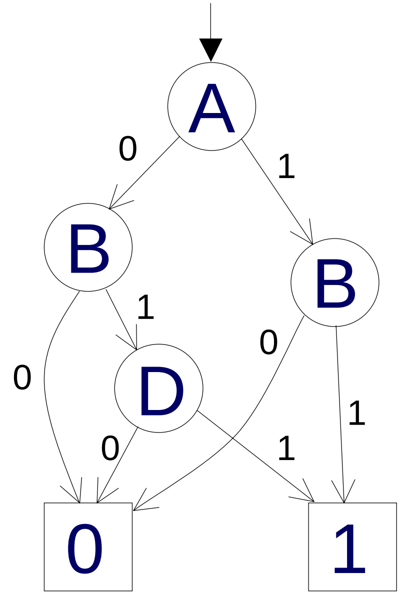
Bryant 1986

Worst case: no improvement - Typical case: indispensable!

Binary Decision Diagrams (BDDs)

A **Binary Decision Diagram** is a canonical graph representation for boolean functions $\{0,1\}^k \rightarrow \{0,1\}$

Example:



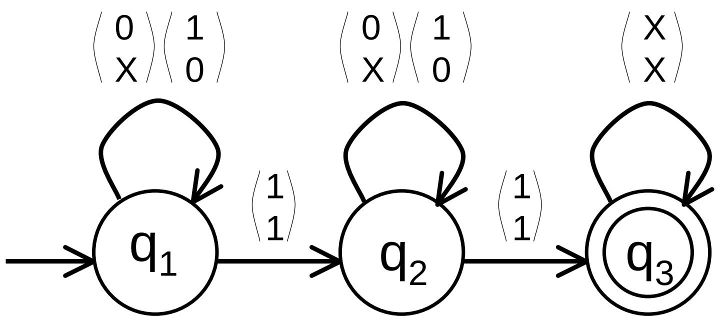
[A=0, B=1, C=0, D=1] is mapped to 1

Binary Decision Diagrams (BDDs)

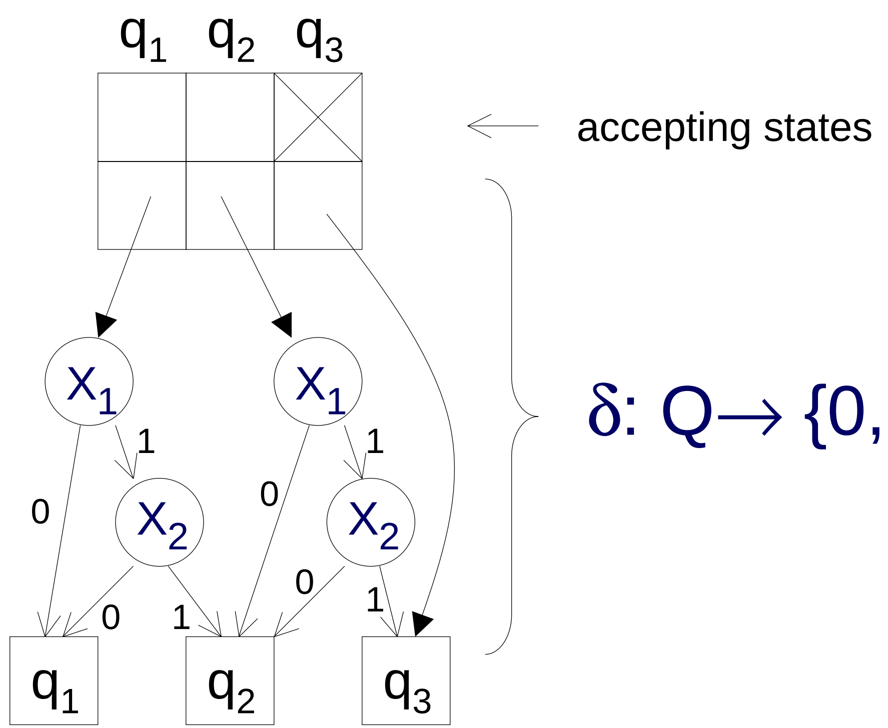
We use **Shared Multi-terminal** BDDs:

- *Shared*: each node represents a function
- *Multi-terminal*: the leaves are $\{q_1, q_2, \ldots, q_n\}$ (not just $\{0,1\}$)

<u>*conventional representation*</u>



<u>*MONA representation*</u>



$$\delta: Q \to \{0,1\}^k \to Q$$

DAGification

Use BDD properties to reuse computations:

Example:

In the formula $\exists X_7 : X_1 \subseteq X_7 \wedge X_7 \subseteq X_9$

the automata for $X_1 \subseteq X_7$ and $X_7 \subseteq X_9$ are isomorphic

DAGification:

- Collapse the formula parse tree to a DAG where the edges are labeled with renaming information
- Build only one automaton for each DAG node

– gives a factor 2–5 speed-up

Formula Reduction

– optimize the formulas before translating into automata

- Simple reductions:

  $\text{true} \vee \Phi \rightarrow \text{true}$, $\neg\neg\,\Phi \rightarrow \Phi$, etc.

- Quantifier reductions: (can give exponential speed-up!)

  $\exists X\!: \Phi \rightarrow \Phi[T/X]$ if $\Phi \Rightarrow X{=}T$ and $FV(T) \subseteq FV(\Phi)$

- Conjunction reductions:

  $\Phi_1 \wedge \Phi_2 \rightarrow \Phi_1$ if Φ_2 is "contained in" Φ_1

– gives a factor 2–4 speed-up

Applications

- Hardware verification [CAV'95, ISMVL'99, FMCAD'00]
- Controller synthesis [FASE'98, FASE'00]
- Trace abstractions [PODC'96]
- Computational linguistics [LACL'97]
- Protocol verification [TACAS'95, FORTE'00]
- Duration calculus
- Parser generation [DLT'99]
- Software engineering [OOPSLA'96]
- Model checking [TACAS'00]
- Theorem proving [CAV'00, FROCOS]
- **Program verification** [PLDI'97, ESOP'00, PLDI'01]
- …

Pointer Assertion Logic [PLDI'01]

Consider an imperative **programming language** for data-type implementations, based on **pointers**

Correctness requirements are specified with **assertions** and **pre/post-conditions**

If

- the assertion language ("Pointer Assertion Logic") is based on **M2L-Tree**,
- the data-types are restricted to certain **tree-like** structures ("graph types" [POPL'93]), and
- the program is sufficiently **annotated**

then correctness can be encoded as MONA formulas!

Red-Black Search Trees

Example: A **red-black search tree** is

1. a binary tree whose records are red or black and have parent pointers
2. a red record cannot have a red successor
3. the root is black
4. the number of black records is the same for all direct paths from the root to a leaf

- 1) is a graph type ☺
- 2) and 3) can be captured as PAL formulas ☺
- 4) cannot be expressed ☹

The `redblackinsert` procedure

```
proc redblackinsert(data t,root:Node):Node  [t.left=null & t.right=null & inv(root)]
{ pointer y,x:Node;
  x = t;
  root = treeinsert(x,root) [treeinsert.Z=x & treeinsert.Q=root];
  x.color = false;
  while [x!=null & root<(left+right)*>x & almostinv1(root,x) & (black(root) | x=root) & (x!=root & red(x.p) => red(x))]
        (x!=root & x.p.color=false) {
    if (x.p=x.p.p.left) {
      y = x.p.p.right;
      if (y!=null & y.color=false) {
       x.p.color = true;
       y.color = true;
       x.p.p.color = false;
       x = x.p.p;
      }
      else {
       if (x=x.p.right) {
         x = x.p;
         root = leftrotate(x,root) [leftrotate.X=x & root<(left+right)*>x & red(leftrotate.Y)];
       }
       x.p.color = true;
       x.p.p.color = false;
       root = rightrotate(x.p.p,root) [rightrotate.Y.left=x & root<(left+right)*>x &
                                       red(rightrotate.X) & rightrotate.Q=root & x!=null];
       root.color = true;
    } }
    else { ... }}
  root.color = true;
  return root;
}  [inv(return)]
```

+ auxiliary procedures `leftrotate`, `rightrotate`, and `treeinsert` (total ~135 lines of program code)

Hoare Logic

1. Require **invariants** at all while-loops and procedure calls (extra assertions are also allowed)

2. Split the program into **Hoare triples**: $\{\Phi_{\mathrm{pre}}\}$ *stm* $\{\Phi_{\mathrm{post}}\}$

3. Verify each triple separately (only loop-free code left)
   - including check for null-pointer dereferences and other memory errors

Note: highly modular, no fixed-point iteration, but requires invariants!

Verifying the Hoare triples

Use a technique of ***transductions*** [CAAP'94] to encode loop-free code:

- A collection of M2L-Tree ***store predicates*** describes a set of stores at a given program point, e.g:
  - succ_T_d(v,w) is true if v denotes a record of type T with a pointer field d pointing to the record w
  - ptr_p(v) is true if v denotes the record pointed to by the program variable p
- Each statement is simulated by ***predicate transformation***, e.g:

  p = q.next;

  is simulated by updating the ptr_p(v) predicate to

  ptr_p'(v) = $\exists$w. ptr_q(w) $\wedge$ succ_T_next(w,v)
- A ***verification condition*** is constructed by expressing the pre- and post-condition using store predicates from end points

This technique is sound *and complete* for individual Hoare triples!

Pointer Assertion Logic

PALE: The Pointer Assertion Logic Engine

- an implementation of this program verification technique

`redblackinsert` ~ 800K formulas

Result of running PALE on **`redblackinsert`**:

After ~4000 tree automaton operations and 40 seconds, PALE replies that

- **all assertions are valid**
- there can be **no null-pointer dereferences or memory leaks**
- the graph type is **wellformed** and **valid** at all cut-points

If verification fails, a **counterexample** initial store is returned

Conclusion

MONA v1.4:

- implementation of classical **logic/automaton** theories
- orders of magnitude more **efficient** than the first implementation due to BDDs, formula reductions, etc.

Future plans:

- heuristic optimizations
- high-level language extensions
- more applications

More information:

The MONA Project: **`http://www.brics.dk/mona/`**

Pointer Assertion Logic: **`http://www.brics.dk/PALE/`**

(Open Source implementations, full documentation, papers, ...)