Discussion:

- Originally, BDDs have been developed for circuit verification.
- Today, they are also applied to the verification of software ...
- A system state is encoded by a sequence of bits.
- A BDD then describes the set of all reachable system states.
- **Warning:** Repeated application of Boolean operations may increase the size dramatically!
- The variable ordering may have a dramatic impact ...
Example: \((x_1 \leftrightarrow x_2) \land (x_3 \leftrightarrow x_4)\)
Discussion (2):

• In general, consider the function:

\[(x_1 \leftrightarrow x_2) \land \ldots \land (x_{2n-1} \leftrightarrow x_{2n})\]

W.r.t. the variable ordering:

\[x_1 < x_2 < \ldots < x_{2n}\]

the BDD has \(3n\) internal nodes.

W.r.t. the variable ordering:

\[x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}\]

the BDD has more than \(2^n\) internal nodes !!

• A similar result holds for the implementation of Addition through BDDs.
Discussion (3):

- Not all Boolean functions have small BDDs :-(
- Difficult functions:
  - multiplication;
  - indirect addressing ...

⇒ data-intensive programs cannot be analyzed in this way :-(

Perspectives: Further Properties of Programs

Freeness: Is $X_i$ possibly/always unbound?

$\implies$

- If $X_i$ is always unbound, no indexing for $X_i$ is required :-(
- If $X_i$ is never unbound, indexing for $X_i$ is complete :-(

Pair Sharing: Are $X_i, X_j$ possibly bound to terms $t_i, t_j$ with

$$\text{Vars}(t_i) \cap \text{Vars}(t_j) \neq \emptyset$$

$\implies$

- Literals without sharing can be executed in parallel :-(

Remark:

Both analyses may profit from **Groundness**!
5.2 Types for Prolog

Example:

\[
\begin{align*}
\text{nat}(X) & \leftarrow X = 0 \\
\text{nat}(X) & \leftarrow X = s(Y), \text{nat}(Y) \\
\text{nat\_list}(X) & \leftarrow X = [] \\
\text{nat\_list}(X) & \leftarrow X = [H|T], \text{nat}(H), \text{nat\_list}(T')
\end{align*}
\]
Discussion

- In Prolog, a type is a set of ground terms with a simple description.
- There is no common agreement what simple means  :-)
- One possibility are (non-deterministic) finite tree automata or normal Horn clauses:

```prolog
nat_list([H|T]) ← nat(H), nat_list(T)  normal
bin(node(T, T)) ← bin(T)  nicht normal
tree(node(T₁, T₂)) ← tree(T₁), tree(T₂)  normal
```
Comparison:

<table>
<thead>
<tr>
<th>Normal clauses</th>
<th>Tree automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary predicate</td>
<td>state</td>
</tr>
<tr>
<td>normal clause</td>
<td>transition</td>
</tr>
<tr>
<td>constructor in the head body</td>
<td>input symbol</td>
</tr>
<tr>
<td>body</td>
<td>pre-condition</td>
</tr>
</tbody>
</table>

General Form:

\[
p(a(X_1, \ldots, X_k)) \leftarrow p_1(X_1), \ldots, p_k(X_k)
\]

\[
p(X) \leftarrow
\]

\[
p(b) \leftarrow
\]
Properties:

- Types then are in fact regular tree languages
- Types are closed under intersection:
  \[
  \langle p, q \rangle(a(X_1, \ldots, X_k)) \leftarrow \langle p_1, q_1 \rangle(X_1), \ldots, \langle p_k, q_k \rangle(X_k) \quad \text{if}
  \]
  \[
  p(a(X_1, \ldots, X_k)) \leftarrow p_1(X_1), \ldots, p_k(X_k)
  \]
  \[
  q(a(X_1, \ldots, X_k)) \leftarrow q_1(X_1), \ldots, q_k(X_k)
  \]
  and

- Types are also closed under union
- Queries \( p(X) \) and \( p(t) \) can be decided in polynomial time but:
  ... only in presence of tabulation!

- Or the program is topdown deterministic ...
Example: Topdown vs. Bottom-up

\[ p(a(X_1, X_2)) \leftarrow p_1(X_1), p_2(X_2) \]
\[ p(a(X_1, X_2)) \leftarrow p_2(X_1), p_1(X_2) \]
\[ p_1(b) \leftarrow \]
\[ p_2(c) \leftarrow \]

... is bottom-up, but not topdown deterministic.

There is no topdown deterministic program for this type!

⇒⇒⇒

Topdown deterministic types are closed under intersection, but not under union !!!
For a set $T$ of terms, we define the set $\Pi(T)$ of paths in terms from $T$:

\[\Pi(T) = \bigcup \{\Pi(t) \mid t \in T\}\]

\[\Pi(b) = \{b\}\]

\[\Pi(a(t_1, \ldots, t_k)) = \{a_j w \mid w \in \Pi(t_j)\} \quad (k > 0)\]

// for new unary constructors $a_j$

Example

\[T = \{a(b, c), a(c, b)\}\]

\[\Pi(T) = \{a_1 b, a_2 c, a_1 c, a_2 b\}\]
Vice versa from a set $P$ of paths, a set $\Pi^{-}(P)$ of terms can be recovered:

$$\Pi^{-}(P) = \{ t \mid \Pi(t) \subseteq P \}$$

**Example (Cont.):**

$$P = \{ a_1 b, a_2 c, a_1 c, a_2 b \}$$

$$\Pi^{-}(P) = \{ a(b, b), a(b, c), a(c, b), a(c, c) \}$$

The set has become larger !!
Theorem:

Assume that $T$ is a regular set of terms. Then:

- $\Pi(T)$ is regular.
- $T \subseteq \Pi^-(\Pi(T))$.
- $T = \Pi^-(\Pi(T))$ iff $T$ is topdown deterministic.
- $\Pi^-(\Pi(T))$ is the smallest superset of $T$ which is topdown deterministic.

Consequence:

If we are interested in topdown deterministic types, it suffices to determine the set of paths in terms.
Example (Cont.):

\[
\begin{align*}
\text{add}(X, Y, Z) & \leftarrow X = 0, \text{nat}(Y), Y = Z \\
\text{add}(X, Y, Z) & \leftarrow \text{nat}(X), X = s(X'), Z = s(Z'), \text{add}(X', Y, Z') \\
\text{mult}(X, Y, Z) & \leftarrow X = 0, \text{nat}(Y), Z = 0 \\
\text{mult}(X, Y, Z) & \leftarrow \text{nat}(X), X = s(X'), \text{mult}(X', Y, Z'), \text{add}(Z', Y, Z)
\end{align*}
\]

Question:
Which run-time checks are necessary?
Idea:

- Approximate the semantics of predicates by means of toptdown-deterministic regular tree languages!
- Alternatively: Approximate the set of paths in the semantics of predicates by regular word languages!

Idea:

- All predicates $p/k$, $k > 0$, are split into predicates $p_1/1, \ldots, p_k/1$. 
Semantics:

Let $\mathcal{C}$ denote a set of clauses.

The set $\left[p\right]_\mathcal{C}$ is the set of tuples of ground terms $(s_1, \ldots, s_k)$, for which $p(s_1, \ldots, s_k)$ is provable.

$\left[p\right]_\mathcal{C}$ (p predicate) thus is the smallest collection of sets of tuples for which:

$$\sigma(t) \in \left[p\right]_\mathcal{C} \quad \text{when ever} \quad \forall i. \sigma(t_i) \in \left[p_i\right]_\mathcal{C}$$

for clauses $p(t) \leftarrow p_1(t_1), \ldots, p_n(t_n) \in \mathcal{C}$ and ground substitutions $\sigma$. 
Approximation of Paths:

Every clause

\[ p(t_1, \ldots, t_k) \leftarrow \alpha \]

is approximated by the clauses:

\[ p_j(w) \leftarrow \bigwedge \Pi(\alpha) \quad \text{where} \]

\[ \Pi(g_1, \ldots, g_m) = \Pi(g_1) \cup \ldots \cup \Pi(g_m) \]

\[ \Pi(q(s_1, \ldots, s_n)) = \{ q_i(w) \mid w \in \Pi(s_i) \} \]

\((j = 1, \ldots, k, w \in \Pi(t_j))\).

Example:

\[ \text{add}(0, Y, Y) \leftarrow \text{nat}(Y) \]

\[ \text{add}(s(X), Y, s(Z)) \leftarrow \text{add}(X, Y, Z) \]
yields:

\[
\begin{align*}
\text{add}_1(0) & \leftarrow \text{nat}_1(Y) \\
\text{add}_2(Y) & \leftarrow \text{nat}_1(Y) \\
\text{add}_3(Y) & \leftarrow \text{nat}_1(Y) \\
\text{add}_1(s_1 X) & \leftarrow \text{add}_1(X), \text{add}_2(Y), \\
& \quad \text{add}_3(Z) \\
\text{add}_2(Y) & \leftarrow \text{add}_1(X), \text{add}_2(Y), \\
& \quad \text{add}_3(Z) \\
\text{add}_3(s_1 Z) & \leftarrow \text{add}_1(X), \text{add}_2(Y), \\
& \quad \text{add}_3(Z)
\end{align*}
\]
Discussion:

• Every literal has at most one occurrence of a variable.

• The literals \( q_j(w_jY) \) where the variable \( Y \) does not occur in the head, represent tests:

  If there is a \( w \) with \( w_jw \in [q_j]_C \) for all such \( j \), then we can cancel these literals.

  If there is no such \( w \), then we can cancel the clause ...

... in the Example:

The literals:

\[
\text{add}_1(X), \text{add}_2(Y), \text{add}_3(Z)
\]

are all satisfiable \( :-) \)
We conclude:

\[
\begin{align*}
\text{add}_1(0) & \leftarrow \\
\text{add}_2(Y) & \leftarrow \text{nat}_1(Y) \\
\text{add}_3(Y) & \leftarrow \text{nat}_1(Y) \\
\text{add}_1(s_1 X) & \leftarrow \text{add}_1(X) \\
\text{add}_2(Y) & \leftarrow \text{add}_2(Y) \\
\text{add}_3(s_1 Z) & \leftarrow \text{add}_3(Z)
\end{align*}
\]
We conclude:

\[
\begin{align*}
\text{add}_1(0) & \leftarrow \\
\text{add}_2(Y) & \leftarrow \text{nat}_1(Y) \\
\text{add}_3(Y) & \leftarrow \text{nat}_1(Y) \\
\text{add}_1(s_1X) & \leftarrow \text{add}_1(X) \\
\text{add}_3(s_1Z) & \leftarrow \text{add}_3(Z)
\end{align*}
\]
We verify:

**Theorem**

Assume that $C$ is a set of clauses. Let $C\#$ denote the corresponding set of clauses for the paths. Then for all predicates $p/k$:

$$\Pi([p]_C) \subseteq [p_1]_{C\#} \cup \ldots \cup [p_k]_{C\#}$$

**Proof:**

Induction on the approximations of the respective fixpoints  

:-)
A set of clauses with unary predicates and unary constructors is called **Alternating Pushdown System** (APS).

**Theorem**

- Every APS is equivalent to a *simple* APS of the form:

  \[
  p(a \, X) \leftarrow p_1(X), \ldots, p_r(X) \\
  p(X) \leftarrow \\
  p(b) \leftarrow 
  \]

- Every APS is equivalent to a normal APS of the form:

  \[
  p(a \, X) \leftarrow p_1(X) \\
  p(X) \leftarrow \\
  p(b) \leftarrow 
  \]
Step 1: Removal of complicated heads:

For \( w = a^{(1)} \ldots a^{(m)} \) \((m > 1)\) we replace

\[
\begin{align*}
    p(w \ X) & \leftarrow \text{rhs} \\
    p(a^{(1)} \ X) & \leftarrow p_2(X) \\
    p_2(a^{(2)} \ X) & \leftarrow p_3(X) \\
    & \ldots \\
    p_{m-1}(a^{(m-1)} \ X) & \leftarrow p_m(X) \\
    p_m(a^{(m)} \ X) & \leftarrow \text{rhs} \\
    & \text{all new}
\end{align*}
\]
Step 1 (Cont.): Removal of complicated heads:

For  \( w = a^{(1)} \ldots a^{(m)}b \)  \( (m > 0) \)  we replace

\[
\begin{align*}
  p(w) & \leftarrow \text{rhs} \\
  p(a^{(1)} X) & \leftarrow p_2(X) \\
  p_2(a^{(2)} X) & \leftarrow p_3(X) \\
  \ldots & \\
  p_{m-1}(a^{(m-1)} X) & \leftarrow p_m(X) \\
  p_m(a^{(m)} X) & \leftarrow p_{m+1}(X) \\
  p_{m+1}(b) & \leftarrow \text{rhs} \\
  // & p_j \text{ all new}
\end{align*}
\]
Step 2: Splitting

We separate independent parts of pre-conditions into auxiliary predicates:

\[
\text{head} \leftarrow \text{rest}, p_1(w_1 X), \ldots, p_m(w_m X)
\]

\((X \text{ does not occur in head, rest})\)

is replaced with:

\[
\text{head} \leftarrow \text{rest}, q()
\]

\[
q() \leftarrow p_1(w_1 X), \ldots, p_m(w_m X)
\]

for a new predicate \(q/0\).