Example (Cont.):

Furthermore,

$$\begin{array}{lll} [\![\mathsf{app}]\!]^\sharp(Z) & \sqsupseteq & X \wedge Y \wedge Z \\ [\![\mathsf{app}]\!]^\sharp(Z) & \sqsupseteq & \textbf{let}\ \psi = X \wedge H \wedge X' \wedge Z \wedge Z' \\ & & \textbf{in}\ \ \exists\, H, X', Z'.\ \mathsf{combine}^\sharp_{\dots}\,(\psi, [\![\mathsf{app}]\!]^\sharp(\mathsf{enter}^\sharp_{\dots}\,(\psi))) \end{array}$$

where for $\psi = Z \wedge H \wedge Z' \wedge (X \leftrightarrow X')$:

$$\begin{array}{lll} \mathsf{enter}^\sharp_{\dots}(\psi) & = & Z \\ \mathsf{combine}^\sharp_{\dots}(\psi, X \wedge Y \wedge Z) & = & X \wedge H \wedge X' \wedge Y \wedge Z \wedge Z' \end{array}$$

Fixpoint iteration therefore yields:

$$[\![\mathsf{app}]\!]^\sharp\,(X) \;=\; X \wedge (Y \leftrightarrow Z) \qquad [\![\mathsf{app}]\!]^\sharp\,(Z) \;=\; X \wedge Y \wedge Z$$

Discussion:

- Exhaustive tabulation of the transformation $[\![\mathsf{app}]\!]^\sharp$ is not feasible.
- Therefore, we rely on demand-driven fixpoint iteration !
- The evaluation starts with the evaluation of the query g, i.e., with the evaluation of $[\![g]\!]^\sharp\,1$.
- The set of inspected fixpoint variables $[\![p]\!]^\sharp\,\psi$ yields a description of all possible calls :-))
- For an efficient representation of functions $\psi \in \mathsf{Pos}$ we rely on binary decision diagrams (BDDs).

Background 6: Binary Decision Diagrams

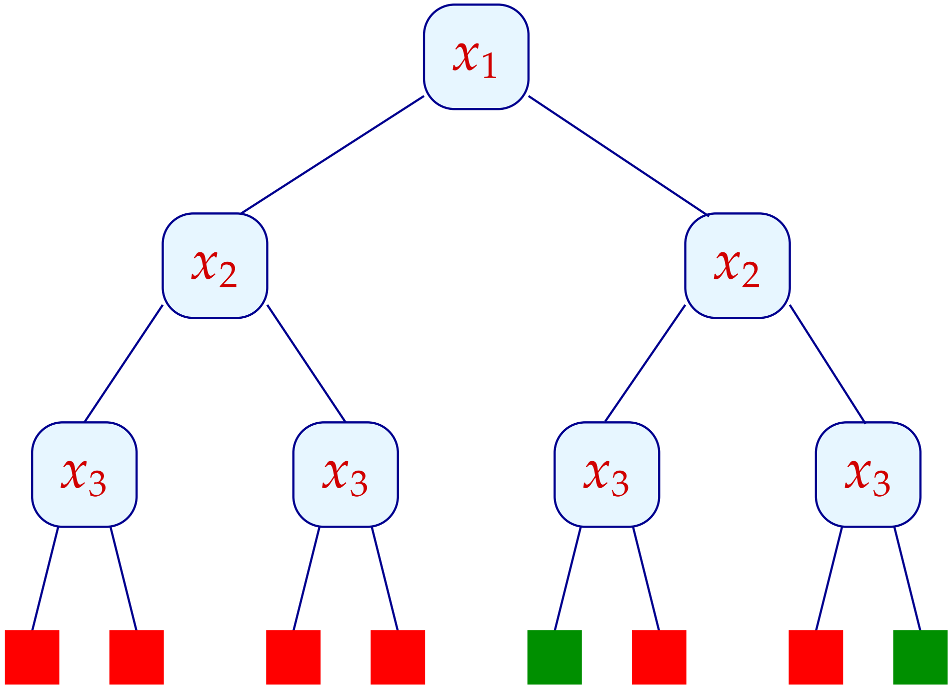
Idea (1):

- Choose an ordering $x_1, \ldots, x_k$ on the arguments ...
- Represent the function $f : \mathbb{B} \to \ldots \to \mathbb{B}$ by $[f]_0$ where:

$$\begin{array}{lll} [b]_k & = & b \\ [f]_{i-1} & = & \textbf{fun } x_i \;\to\; \textbf{if } x_i \textbf{ then } [f\ 1]_i \\ & & \qquad\qquad\quad \textbf{else } [f\ 0]_i \end{array}$$

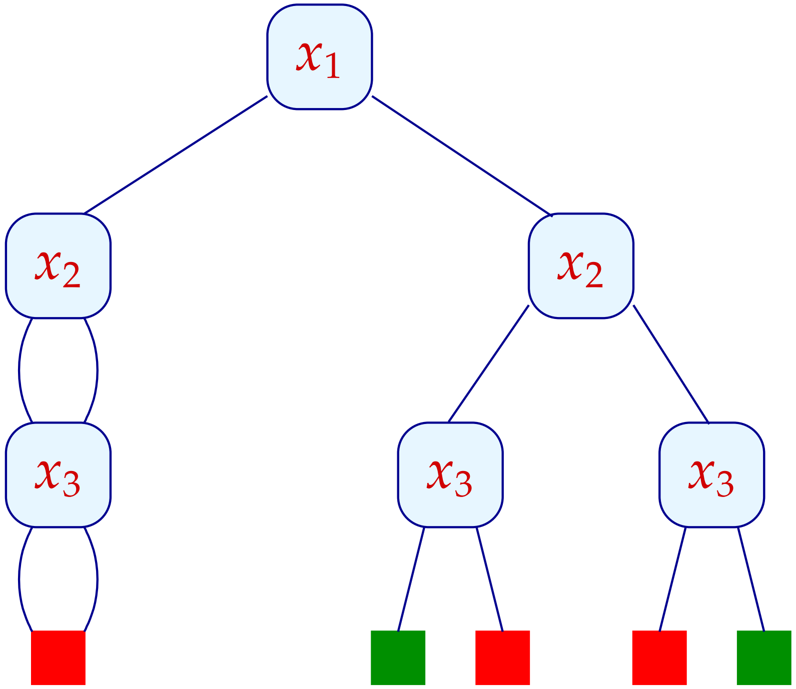
Example: $f\ x_1\ x_2\ x_3 \;=\; x_1 \wedge (x_2 \leftrightarrow x_3)$

... yields the tree:



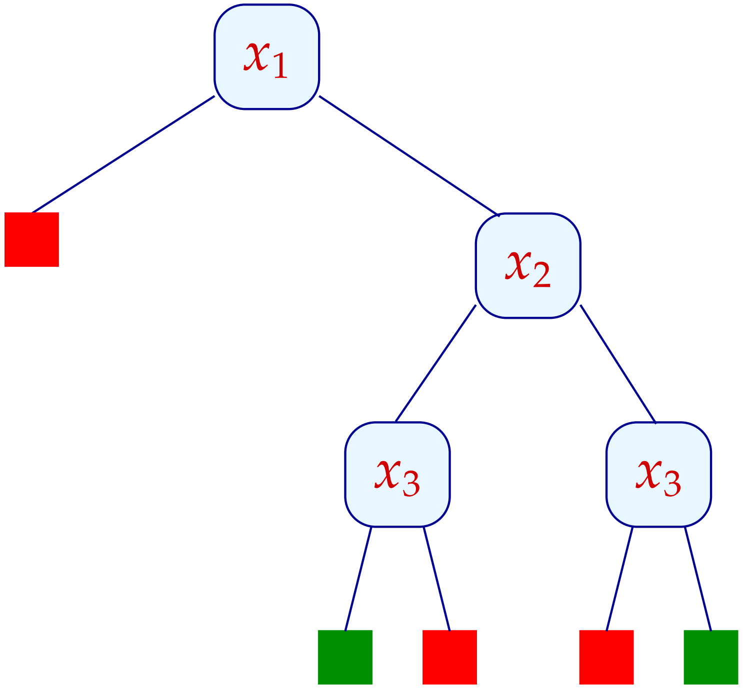
Idea (2):

- Decision trees are exponentially large :-(
- Often, however, many sub-trees are isomorphic :-)
- Isomorphic sub-trees need to be represented only once ...



Idea (3):

- Nodes whose test is irrelevant, can also be abandoned ...



Discussion:

- This representation of the Boolean function f is unique !

  $\Longrightarrow$

  Equality of functions is efficiently decidable !!
- For the representation to be useful, it should support the basic operations: $\wedge, \vee, \neg, \Rightarrow, \exists\, x_j$ ...

$$
\begin{array}{lcl}
[b_1 \wedge b_2]_k & = & b_1 \wedge b_2 \\
[f \wedge g]_{i-1} & = & \textbf{fun } x_i \ \rightarrow \ \textbf{if } x_i \textbf{ then } [f\,1 \wedge g\,1]_i \\
 & & \qquad\qquad\quad \textbf{else } [f\,0 \wedge g\,0]_i \\
 & & // \quad \text{analogous for the remaining operators}
\end{array}
$$

$$
\begin{array}{lcll}
[\exists\, x_j.\, f]_{i-1} & = & \textbf{fun } x_i \;\rightarrow\; \textbf{if } x_i \textbf{ then } [\exists\, x_j.\, f\; 1]_i & \\
 & & \qquad\qquad \textbf{else } [\exists\, x_j.\, f\; 0]_i & \text{if } i < j \\
[\exists\, x_j.\, f]_{j-1} & = & [f\; 0 \vee f\; 1]_j &
\end{array}
$$

- Operations are executed bottom-up.
- Root nodes of already constructed sub-graphs are stored in a unique-table

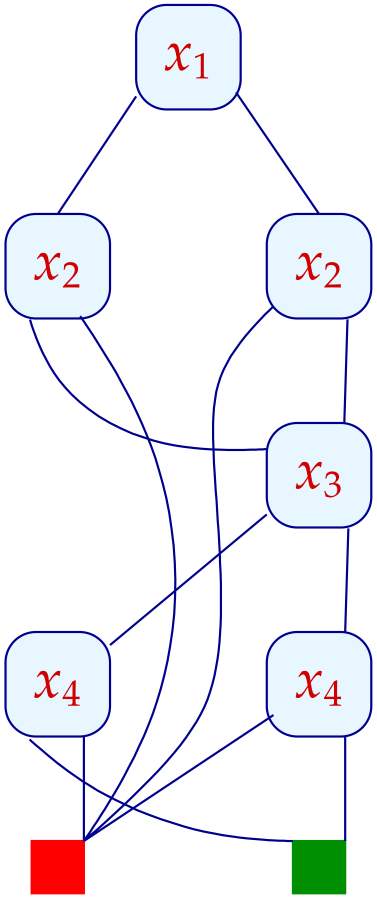
  $\Longrightarrow$

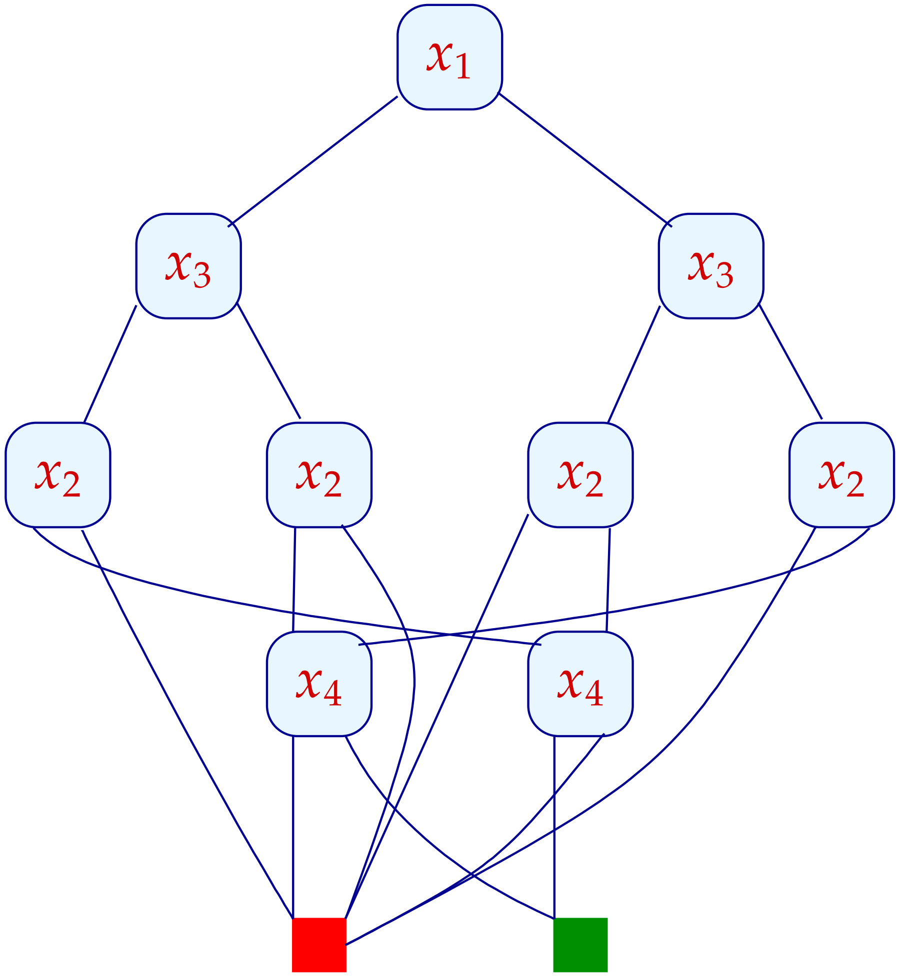
  Isomorphy can be tested in constant time !
- The operations thus are polynomial in the size of the input BDDs :-)

Discussion:

- Originally, BDDs have been developped 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: $(x_1 \leftrightarrow x_2) \wedge (x_3 \leftrightarrow x_4)$





Discussion (2):

- In general, consider the function:

$$(x_1 \leftrightarrow x_2) \wedge \ldots \wedge (x_{2n-1} \leftrightarrow x_{2n})$$

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 ...

$\Longrightarrow$ data-intensive programs cannot be analyzed in this way :-(

Perspectives: Further Properties of Programs

Freeness: Is X_i possibly/always unbound ?

$\Longrightarrow$

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

$$Vars(t_i) \cap Vars(t_j) \neq \emptyset \quad ?$$

$\Longrightarrow$

Literals without sharing can be executed in parallel :-)

Remark:

Both analyses may profit from **Groundness** !

5.2 Types for Prolog

Example:

$$\begin{array}{lcl} \mathsf{nat}(X) & \leftarrow & X = 0 \\ \mathsf{nat}(X) & \leftarrow & X = s(Y), \mathsf{nat}(Y) \\ \mathsf{nat\_list}(X) & \leftarrow & X = [] \\ \mathsf{nat\_list}(X) & \leftarrow & X = [H|T], \mathsf{nat}(H), \mathsf{nat\_list}(T) \end{array}$$

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:

$$\begin{array}{lcll}
\mathsf{nat\_list}([H|T]) & \leftarrow & \mathsf{nat}(H), \mathsf{nat\_list}(T) & \text{normal} \\
\mathsf{bin}(node(T,T)) & \leftarrow & \mathsf{bin}(T) & \text{nicht normal} \\
\mathsf{tree}(node(T_1,T_2)) & \leftarrow & \mathsf{tree}(T_1), \mathsf{tree}(T_2) & \text{normal}
\end{array}$$

Comparison:

Normal clauses	Tree automaton
unary predicate	state
normal clause	transition
constructor in the head	input symbol
body	pre-condition

General Form:

$$\begin{array}{lcl} p(a(X_1,\ldots,X_k)) & \leftarrow & p_1(X_1),\ldots,p_k(X_k) \\ p(X) & \leftarrow & \\ p(b) & \leftarrow & \end{array}$$

Properties:

- Types then are in fact regular tree languages ;-)
- Types are closed under intersection:

$$
\begin{array}{lcll}
\langle p, q\rangle(a(X_1,\ldots,X_k)) & \leftarrow & \langle p_1, q_1\rangle(X_1),\ldots,\langle p_k, q_k\rangle(X_k) & \text{if} \\
p(a(X_1,\ldots,X_k)) & \leftarrow & p_1(X_1),\ldots,p_k(X_k) & \text{and} \\
q(a(X_1,\ldots,X_k)) & \leftarrow & q_1(X_1),\ldots,q_k(X_k) &
\end{array}
$$

- 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

$$\begin{array}{lcl} p(a(X_1, X_2)) & \leftarrow & p_1(X_1), p_2(X_2) \\ p(a(X_1, X_2)) & \leftarrow & p_2(X_1), p_1(X_2) \\ p_1(b) & \leftarrow & \\ p_2(c) & \leftarrow & \end{array}$$

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

There is no topdown deterministic program for this type !

$\Longrightarrow$

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:

$$
\begin{array}{lcl}
\Pi(T) & = & \bigcup\{\Pi(t) \mid t \in T\} \\
\\
\Pi(b) & = & \{b\} \\
\Pi(a(t_1, \ldots, t_k)) & = & \{a_j w \mid w \in \Pi(t_j)\} \qquad (k > 0) \\
 & & // \quad \text{for new unary constructors } a_j
\end{array}
$$

Example

$$
\begin{array}{lcl}
T & = & \{a(b,c), a(c,b)\} \\
\Pi(T) & = & \{a_1 b, a_2 c, a_1 c, a_2 b\}
\end{array}
$$

Vice versa from a set P of paths, a set $\Pi^-(P)$ of terms can be recovered:

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

Example (Cont.):

$$\begin{array}{lll} P & = & \{a_1 b, a_2 c, a_1 c, a_2 b\} \\ \Pi^-(P) & = & \{a(b,b), a(b,c), a(c,b), a(c,c)\} \end{array}$$

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.):

$$
\begin{array}{lcl}
\mathsf{add}(X,Y,Z) & \leftarrow & X = 0, \mathsf{nat}(Y), Y = Z \\
\mathsf{add}(X,Y,Z) & \leftarrow & \mathsf{nat}(X), X = s(X'), Z = s(Z'), \mathsf{add}(X',Y,Z') \\
\mathsf{mult}(X,Y,Z) & \leftarrow & X = 0, \mathsf{nat}(Y), Z = 0 \\
\mathsf{mult}(X,Y,Z) & \leftarrow & \mathsf{nat}(X), X = s(X'), \mathsf{mult}(X',Y,Z'), \mathsf{add}(Z',Y,Z)
\end{array}
$$

Question:

Which run-time checks are necessary?

Idea:

- Approximate the semantics of predicates by means of topdown-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 $[\![p]\!]_{\mathcal{C}}$ is the set of tuples of ground terms $(s_1, \ldots, s_k)$, for which $p(s_1, \ldots, s_k)$ is provable :-)

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

$$\sigma(\underline{t}) \in [\![p]\!]_{\mathcal{C}} \quad \text{when ever} \quad \forall\, i.\ \sigma(\underline{t}_i) \in [\![p_i]\!]_{\mathcal{C}}$$

for clauses $p(\underline{t}) \leftarrow p_1(\underline{t}_1), \ldots, p_n(\underline{t}_n) \in \mathcal{C}$ and ground substitutions σ.