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 ...

---

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 ...

$$\implies$$ 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!

Example:

```
biggest(X, Y) ← X = elephant, Y = horse
biggest(X, Y) ← X = horse, Y = donkey
biggest(X, Y) ← X = donkey, Y = dog
biggest(X, Y) ← X = donkey, Y = monkey
is_biggest(X, Y) ← biggest(X, Y)
is_biggest(X, Y) ← biggest(X, Z), is_biggest(Z, Y)
              ← is_biggest(elephant, dog)
```

Example:

```
biggest(X, Y) ← X = elephant, Y = horse
biggest(X, Y) ← X = horse, Y = donkey
biggest(X, Y) ← X = donkey, Y = dog
biggest(X, Y) ← X = donkey, Y = monkey
is_biggest(X, Y) ← biggest(X, Y)
is_biggest(X, Y) ← biggest(X, Z), is_biggest(Z, Y)
              ← is_biggest(elephant, dog)
```
A more realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H, [X']], Z = [H', Z'], \text{app}(X', Y, Z') \\
& \leftarrow \text{app}(X, [Y, c], [a, b, Z])
\end{align*}
\]

\[
\begin{align*}
X &= a \\
Y &= b \\
Z &= c
\end{align*}
\]

Remark:

[] \quad \text{the atom empty list} \\
[H|Z] \quad \text{binary constructor application} \\
[a, b, Z] \quad \text{Abbreviation for: } \langle a|b|[Z]|\rangle

Accordingly, a program \( p \) is constructed as follows:

\[
\begin{align*}
t & ::= a \mid X \mid f(t_1, \ldots, t_n) \\
g & ::= p(t_1, \ldots, t_k) \mid X = t \\
c & ::= p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
q & ::= g_1, \ldots, g_r \\
p & ::= c_1 \cdots c_m q
\end{align*}
\]

- A term \( t \) either is an atom, a (possibly anonymous) variable or a constructor application.
- A goal \( g \) either is a literal, i.e., a predicate call, or a unification.
- A clause \( c \) consists of a head \( p(X_1, \ldots, X_k) \) together with body consisting of a sequence of goals.
- A program \( p \) consists of a sequence of clauses together with a sequence of goals as query.

Procedural View of PuP-Programs:

- literal \quad \text{procedure call} \\
- predicate \quad \text{procedure} \\
- definition \quad \text{body} \\
- term \quad \text{value} \\
- unification \quad \text{basic computation step} \\
- binding of variables \quad \text{side effect}

Warning:

- do not return results!
- modify the caller solely through side effects \( \implies \) \\
- may fail. Then, the following definition is tried \( \implies \) backtracking
Inefficiencies:

**Backtracking:**  • The matching alternative must be searched for
     ➔ Indexing
  • Since a successful call may still fail later, the stack can only be
cleared if there are no pending alternatives.

**Unification:**  • The translation possibly must switch between build
     and check several times.
  • In case of unification with a variable, an Occur Check must be
    performed.

**Type Checking:**  • Since Prolog is untyped, it must be checked at
     run-time whether or not a term is of the desired form.
  • Otherwise, ugly errors could show up.

A more realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H \mid X'], Z = [H \mid Z'], \text{app}(X', Y, Z') \\
& \leftarrow \text{app}(X, [Y, c], [a, b, Z])
\end{align*}
\]

Remark:

\[
\begin{align*}
[] & \quad \text{the atom empty list} \\
[H \mid Z] & \quad \text{binary constructor application} \\
[a, b, Z] & \quad \text{Abbreviation for: } [a][b[Z] \mid \ ]]
\end{align*}
\]

Some Optimizations:

• Replacing last calls with jumps;
• Compile-time type inference;
• Identification of deterministic predicates ...

Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H \mid X'], Z = [H \mid Z'], \text{app}(X', Y, Z') \\
& \leftarrow \text{app}(a, b, [Y, c], Z)
\end{align*}
\]
Some Optimizations:

- Replacing last calls with jumps;
- Compile-time type inference;
- Identification of deterministic predicates ...

Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z') \\
& \leftarrow \text{app}([a, b], [Y, c], Z)
\end{align*}
\]

5.1 Groundness Analysis

A variable \( X \) is called ground w.r.t. a program execution \( \pi \) starting program entry and entering a program point \( v \), if \( X \) is bound to a variable-free term.

Goal:

- Find all variables which are ground whenever a particular program point is reached!
- Find all arguments of a predicate which are ground whenever the predicate is called!

Observation:

- In PuP, functions must be simulated through predicates.
- These then have designated input- and output parameters.
- Input parameters are those which are instantiated with a variable-free term whenever the predicate is called.
  These are also called ground.
- In the example, the first parameter of \text{app} is an input parameter.
- Unification with such a parameter can be implemented as pattern matching!
- Then we see that \text{app} in fact is deterministic !!!

Some Optimizations:

- Replacing last calls with jumps;
- Compile-time type inference;
- Identification of deterministic predicates ...

Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = [], Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z') \\
& \leftarrow \text{app}([a, b], [Y, c], Z)
\end{align*}
\]
5.1 Groundness Analysis

A variable \( X \) is called ground w.r.t. a program execution \( \pi \) starting program entry and entering a program point \( v \), if \( X \) is bound to a variable-free term.

Goal:
- Find all variables which are ground whenever a particular program point is reached!
- Find all arguments of a predicate which are ground whenever the predicate is called!

Idea:
- Describe groundness by values from \( \mathbb{B} \):
  - \( 1 \) \( \iff \) variable-free term;
  - \( 0 \) \( \iff \) term which contains variables.
- A set of variable assignments is described by Boolean functions \( \triangleright\!
\begin{align*}
X \leftrightarrow Y & \iff X \text{ is ground iff } Y \text{ is ground.} \\
X \wedge Y & \iff X \text{ and } Y \text{ are ground.}
\end{align*}
\)

Idea (cont.):
- The constant function \( \theta \) denotes an unreachable program point.
- Occurring sets of variable assignments are closed under substitution.
  This means that for every occurring function \( \phi \neq 0 \),
  \[
  \phi(1, \ldots, 1) = 1
  \]
  These functions are called positive.
- The set of all positive functions is called \( \text{Pos} \).
  Ordering: \( \phi_1 \sqsubseteq \phi_2 \iff \phi_1 \Rightarrow \phi_2 \).
- In particular, the least element is \( 0 \) \( \triangleright\!
}
Idea:

- Describe groundness by values from \( B \):
  - 1 \( \iff \) variable-free term;
  - 0 \( \iff \) term which contains variables.
- A set of variable assignments is described by Boolean functions.
  - \( X \leftrightarrow Y \) \( \iff \) \( X \) is ground \( \iff \) \( Y \) is ground.
  - \( X \land Y \) \( \iff \) \( X \) and \( Y \) are ground.

Idea (cont.):

- The constant function 0 denotes an unreachable program point.
- Occurring sets of variable assignments are closed under substitution.
  - This means that for every occurring function \( \phi \neq 0 \):
    \[
    \phi(1, \ldots, 1) = 1
    \]
  - These functions are called positive.
- The set of all positive functions is called \( \text{Pos} \).
  - Ordering: \( \phi_1 \subseteq \phi_2 \) if \( \phi_1 \Rightarrow \phi_2 \).
- In particular, the least element is 0.

Example:

Example diagram:

- Not all positive functions are monotonic!!!
- For \( k \) variables, there are \( 2^{2^{k-1}} + 1 \) many functions.
- The height of the complete lattice is \( 2^k \).
- We construct an interprocedural analysis which for every predicate \( p \) determines a (monotonic) transformation
  \[
  [p]^2 : \text{Pos} \rightarrow \text{Pos}
  \]
- For every clause, \( p(X_1, \ldots, X_k) \implies g_1, \ldots, g_m \) we obtain the constraint:
  \[
  [p]^2 \psi \sqsubseteq \exists X_{k+1}, \ldots, X_m, [g_1]^2 \ldots ([g_m]^2 \psi) \ldots
  \]
  - \( m \) number of clause variables
Example:

![Diagram showing a lattice with variables and operations]

Remarks:

- Not all positive functions are monotonic !!!
- For \( k \) variables, there are \( 2^{2^k} - 1 \) many functions.
- The height of the complete lattice is \( 2^k \).
- We construct an interprocedural analysis which for every predicate \( p \) determines a (monotonic) transformation
  
  \[ [p]^2 : \text{Pos} \rightarrow \text{Pos} \]

- For every clause, \( p(X_1, \ldots, X_k) \Leftarrow g_1, \ldots, g_m \) we obtain the constraint:
  
  \[ [p]^2 \psi \sqsubseteq \exists X_{k+1}, \ldots, X_m. [g_1]^2 \ldots \ldots ([g_m]^2 \psi) \ldots \]

  // \( m \) number of clause variables

Abstract Unification:

\[ [X = t]^2 \psi = \psi \land (X \leftrightarrow X_1 \land \ldots \land X_n) \]

if \( \text{Vars}(t) = \{X_1, \ldots, X_n\} \).

Abstract Literal:

\[ [q(s_1, \ldots, s_k)]^2 \psi = \text{combine}_{i=1}^{n} (\psi, [q[i]^2 (\text{null}(s_i, \psi))] \)

// analogous to procedure call !!

Remarks:

- Not all positive functions are monotonic !!!
- For \( k \) variables, there are \( 2^{2^k} - 1 \) many functions.
- The height of the complete lattice is \( 2^k \).
- We construct an interprocedural analysis which for every predicate \( p \) determines a (monotonic) transformation
  
  \[ [p]^2 : \text{Pos} \rightarrow \text{Pos} \]

- For every clause, \( p(X_1, \ldots, X_k) \Leftarrow g_1, \ldots, g_m \) we obtain the constraint:
  
  \[ [p]^2 \psi \sqsubseteq \exists X_{k+1}, \ldots, X_m. [g_1]^2 \ldots \ldots ([g_m]^2 \psi) \ldots \]

  // \( m \) number of clause variables
Abstract Unification:

\[
[X = t]^x_\psi = \psi \land (X \Rightarrow X_1 \land \ldots \land X_r)
\]

if \( \text{Vars}(t) = \{X_1, \ldots, X_r\} \).

Abstract Literal:

\[
[q(s_1, \ldots, s_k)]^x_\psi = \text{combine}_{s_1, \ldots, s_k}^x(\psi, [q]^x_\psi (\text{enter}_{s_1, \ldots, s_k})^x_\psi))
\]

// analogous to procedure call !!

Thereby:

\[
\text{enter}_{s_1, \ldots, s_k}^x \psi = \text{ren}(\exists X_1, \ldots, X_m. [X_1 = s_1, \ldots, X_k = s_k]^x_\psi)
\]

\[
\text{combine}_{s_1, \ldots, s_k}^x(\psi, \psi_1) = \exists X_1, \ldots, X_r. \psi \land [X_1 = s_1, \ldots, X_k = s_k]^x \text{ren}\phi_1
\]

where

\[
\exists X. \phi = \phi[0/X] \lor \phi[1/X]
\]

\[
\text{ren} \phi = \phi[X_1/X_1, \ldots, X_k/X_k]
\]

\[
\text{ren} \phi = \phi[X_1/X_1, \ldots, X_r/X_r]
\]