DPLL-Style Program Analysis

Leopold Haller

POPL Student Blitz Session
Imprecision in Abstract Interpretation

- Abstract interpretation sound but not complete.

- Incompleteness manifests in **imprecision** during the analysis.

Example: Domain of Intervals
**Imprecisions in the Domain**

<table>
<thead>
<tr>
<th>Imprecision in join</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := *;</td>
</tr>
<tr>
<td>if (x &gt; 5)</td>
</tr>
<tr>
<td>y := -1; → y ∈ [-1, -1], x ∈ [6, ∞]</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>y := 1; → y ∈ [1, 1], x ∈ [-∞, 5]</td>
</tr>
<tr>
<td>assert(y != 0); → y ∈ [-1, 1]</td>
</tr>
</tbody>
</table>

The disjunction $y = 1 \lor y = -1$ cannot be expressed as an interval.
Imprecisions in the Domain

Imprecision in join

\[
x := *; \\
\text{if}(x > 5) \\
\quad y := -1; \quad \rightarrow y \in [-1, -1], x \in [6, \infty] \\
\text{else} \\
\quad y := 1; \quad \rightarrow y \in [1, 1], x \in [-\infty, 5] \\
\text{assert}(y \neq 0); \quad \rightarrow y \in [-1, 1]
\]

The disjunction \( y = 1 \lor y = -1 \) cannot be expressed as an interval.

How can we introduce disjunctions just where we need them?
Consider separately different sets of traces through a program

Think: Case splits in a proof.

Control-flow based trace partitioning

- `x := *`  
  - `[x > 5]`  
  - `[x <= 5]`  
- `y := 1`  
- `y := -1`  
- `assert(y != 0)`  
- `y = -1`  
- `y = 1`
Trace Partitioning

- Consider separately different sets of traces through a program
- Think: Case splits in a proof.

### Control-flow based trace partitioning

- $x := *$
- $x > 5$
- $x \leq 5$
- $y := 1$
- $y := -1$
- $\text{assert}(y \neq 0)$
- $y = 1$
Trace Partitioning

- Consider separately different sets of traces through a program
- Think: Case splits in a proof.

```
x := *
[x > 5]
y := -1
assert(y != 0)
[x <= 5]
y := 1
```

```
y = -1
y = 1
```
The main question is:
The main question is:
How can we find a good partitioning?
The main question is:
How can we find a good partitioning?

just precise enough
abstract enough to be efficient
Clipped fixpoints

Standard analysis

\[ \hat{FP} \equiv \mu X. I \sqcup \hat{F}(X) \]

This may be too imprecise for the reasons mentioned earlier.
Clipped fixpoints

Standard analysis

\[ \hat{FP} \equiv \mu X. I \sqcup \hat{F}(X) \]

This may be too imprecise for the reasons mentioned earlier.

Clippings

Find a set \( a_1, \ldots, a_k \) of abstract elements and compute for each \( 1 \leq i \leq k \)

\[ \hat{FP}_i \equiv \mu X. I \sqcup (\hat{F}(X) \cap a_i) \]

such that each program behaviour is represented in some \( \hat{FP}_i \).

Any checks can be performed on the \( FP_i \) for increased precision.

Clippings are equivalent to a certain class of trace partitionings
Reframed question:
Reframed question:
How do we find these elements $a_1, \ldots, a_k$?

Let’s look at an architecture that’s good at dealing with disjunction
Main phases of the DPLL procedure:

- **Decision**: Assume a value for an undetermined variable.
- **Propagation**: Deduce implied variable values.
- **Learning**: Learn reason for conflict and backtrack.

Use the same architecture for program analysis. Current variable assignment corresponds to clipping.
Main phases of the DPLL procedure:
Main phases of the DPLL procedure:

**Decision**  Assume a value for an undetermined variable
Main phases of the DPLL procedure:

- **Decision**: Assume a value for an undetermined variable
- **Propagation**: Deduce implied variable values
Main phases of the DPLL procedure:

- **Decision**: Assume a value for an undetermined variable
- **Propagation**: Deduce implied variable values
- **Learning**: Learn reason for conflict and backtrack
Main phases of the DPLL procedure:

- **Decision**: Assume a value for an undetermined variable
- **Propagations**: Deduce implied variable values
- **Learning**: Learn reason for conflict and backtrack

Use same architecture for program analysis. Current variable assignment corresponds to clipping.
Initially, \( a = \top \)

Decision

Initially, \( a = \top \)

\[
\begin{array}{c}
\top \\
A_1 & A_2 & A_3 & A_4 \\
B_1 & B_2 & B_3 & B_4 & B_5 \\
C_1 & C_2 & C_3 & C_4 \\
\bot \\
\end{array}
\]
Initially, $a = \top$

$\mu X. \hat{F}(X)$ not safe

Initially, $a = \top$

$\top$

$A_1$ $A_2$ $A_3$ $A_4$

$B_1$ $B_2$ $B_3$ $B_4$ $B_5$

$C_1$ $C_2$ $C_3$ $C_4$

$\bot$
SAT-Style Program Analysis

Decision: refine a

Decision
Decision: refine $a$

$\mu X. (\hat{F}(X) \sqcap A_1)$ not safe
SAT-Style Program Analysis

Decision

\[ A_1 \]
\[ A_2 \]
\[ A_3 \]
\[ A_4 \]
\[ B_1 \]
\[ B_2 \]
\[ B_3 \]
\[ B_4 \]
\[ B_5 \]
\[ C_1 \]
\[ C_2 \]
\[ C_3 \]
\[ C_4 \]
\[ \perp \]

Decision: refine \( a \)

Initially, \( a = \perp \mu X. \hat{F}(X) \) not safe

Decision: refine \( a \mu X. (\hat{F}(X) \ominus A_1) \) not safe

\( A_2 \mu X. (\hat{F}(X) \ominus A_2) \) safe

\( B_2 \mu X. (\hat{F}(X) \ominus B_2) \) safe

Backtrack and continue
Decision: refine a

\( \mu X. (\hat{F}(X) \sqcap B_2) \) safe
SAT-Style Program Analysis

Generalization

Decision: refine $a = \top \mu X$. 

$\hat{F}(X) \not\subseteq A_1$ not safe

Decision: refine $a = \mu X$. 

$\hat{F}(X) \not\subseteq A_2 \wedge B_2$ not safe

$A_2 \mu X \wedge \hat{F}(X) \subseteq A_2 \wedge B_2$ safe

Backtrack and continue
SAT-Style Program Analysis

\[ \mu X.\hat{F}(X) \sqcap A_2 \text{ safe} \]
SAT-Style Program Analysis

Decision: refine $\mu X. \hat{F}(X) \sqcap A_2$ safe

Learning (use for propagation)

Generalization
Initially, $a = \top \mu X$.  

$\hat{F}(X)$ not safe

Decision: refine $a \mu X$.  

$(\hat{F}(X) \land A_1)$ not safe

$A_2 \mu X$.  

$\hat{F}(X) \land A_2$ safe

$A_3 \mu X$.  

$A_4 \mu X$.  

$A_1 \land A_2 \land A_3 \land A_4$ safe

Backtrack and continue
Summary & Application

- Refine domain in a property dependent way by using a DPLL style analysis.
- Application to verification of industrial floating-point programs using value-based partitionings

Thanks for your attention.

Leopold Haller (OUCL)
Summary & Application

- Refine domain in a property dependent way by using a DPLL style analysis.
- Application to verification of industrial floating-point programs using value-based partitionings

Thanks for your attention.