Mathematical Theory
Lattice Definition

• A complete lattice \((L, \sqsubseteq)\) is a set \(L\) with a partial ordering \(\sqsubseteq\):
  - Reflexive: \(1 \sqsubseteq 1\)
  - Transitive: \(l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3\)
  - Anti-symmetric: \(l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2\)

and such that all subsets \(Y\) have least upper bounds \(\sqcup Y\):
  - Upper bound: \(\forall l \in Y: l \sqsubseteq \sqcup Y\)
  - Least: if \(\forall l \in Y: l \sqsubseteq l_0\) then \(\sqcup Y \sqsubseteq l_0\)
Lattice Theory: Implications

- A lattice is non-empty.
- Every lattice $L$ has a unique greatest element $\top \in L$.
- All subsets $Y$ of $L$ have greatest lower bounds $\forall Y \in L$.
- Every lattice $L$ has a unique least element $\bot \in L$. 
Fixed point

A **fixed point** (sometimes shortened to **fixpoint**) of a function is a point that is mapped to itself by the function.

**Definition**

\[ x \in L \text{ is a fixed point of the function } f: L \rightarrow L \text{ if and only if } f(x) = x \]
Knaster–Tarski theorem

**Knaster–Tarski theorem**, named after Bronisław Knaster (1893-1990) and Alfred Tarski (1902-1983)

Let $L$ be a complete lattice and let $f : L \rightarrow L$ be a monotone function. Then the set of fixed points of $f$ in $L$ is also a complete lattice.
Tarski: Implications

- Every monotone function (on a complete lattice) has a fixpoint.
- There is a unique least fixpoint.
- There is a unique greatest fixpoint.
Ascending Chain Condition

A lattice \((L, \sqsubseteq)\) satisfies the **ascending chain condition** if every ascending chain
\(l_0 \sqsubseteq l_1 \sqsubseteq \ldots\)
eventually stabilizes i.e.
\(\exists \ m \in \mathbb{N} : l_m = l_{m+1}\)
Kleene fixpoint theorem

Given a complete lattice \((L, \sqsubseteq)\) satisfying the ascending chain condition, and a monotone function \(f : L \rightarrow L\) the least fixpoint of \(f\) \(\text{lfp}(f)\) can be computed as \(f^k(\bot)\) with some \(\exists \ k \in \mathbb{N}\) where

\[
\begin{align*}
f^0(x) &= \bot \\
f^{n+1}(x) &= f(f^n(x))
\end{align*}
\]
Repetition
**Notation**

`Analysis_o` input to the transfer function

`Analysis_·` output of the transfer function

**Forward**

```
Analysis_·
  ▼
Analysis_o
  ▼
Analysis_·
  ▼
Analysis_·
  ▼
```

**Backward**

```
Analysis_·
  ▼
Analysis_o
  ▼
Analysis_·
  ▼
Analysis_·
  ▼
```
Available Expressions

**kill** and **gen** functions

\[
\begin{align*}
\text{kill}_{\text{AE}}([x \leftarrow a]^\ell) & = \{ a' \in \text{AExp}_* \mid x \in \text{FV}(a') \} \\
\text{kill}_{\text{AE}}([\text{skip}]^\ell) & = \emptyset \\
\text{kill}_{\text{AE}}([b]^\ell) & = \emptyset \\
\text{gen}_{\text{AE}}([x \leftarrow a]^\ell) & = \{ a' \in \text{AExp}(a) \mid x \notin \text{FV}(a') \} \\
\text{gen}_{\text{AE}}([\text{skip}]^\ell) & = \emptyset \\
\text{gen}_{\text{AE}}([b]^\ell) & = \text{AExp}(b)
\end{align*}
\]

data flow equations:  

\[
\begin{align*}
\text{AE}_\circ (\ell) & = \begin{cases}
\emptyset & \text{if } \ell = \text{init}(S_*), \\
\cap \{ \text{AE}_\bullet (\ell') \mid (\ell', \ell) \in \text{flow}(S_*) \} & \text{otherwise}
\end{cases} \\
\text{AE}_\bullet (\ell) & = (\text{AE}_\circ (\ell) \setminus \text{kill}_{\text{AE}}(B^\ell)) \cup \text{gen}_{\text{AE}}(B^\ell)
\end{align*}
\]

where \( B^\ell \in \text{blocks}(S_*) \)
Reaching Definitions

**kill** and **gen** functions

\[
\begin{align*}
\text{kill}_{\text{RD}}([x := a]^\ell) &= \{(x, ?)\} \\
&\cup \{(x, \ell') \mid B^{\ell'} \text{ is an assignment to } x \text{ in } S_\ast\}
\end{align*}
\]

\[
\begin{align*}
\text{kill}_{\text{RD}}([\text{skip}]^\ell) &= \emptyset \\
\text{kill}_{\text{RD}}([b]^\ell) &= \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{gen}_{\text{RD}}([x := a]^\ell) &= \{(x, \ell)\} \\
\text{gen}_{\text{RD}}([\text{skip}]^\ell) &= \emptyset \\
\text{gen}_{\text{RD}}([b]^\ell) &= \emptyset
\end{align*}
\]

Data flow equations: $\text{RD}^\text{RD}$

\[
\begin{align*}
\text{RD}_\circ (\ell) &= \begin{cases} 
\{(x, ?) \mid x \in \text{FV}(S_\ast)\} \\
\cup \text{RD}_\bullet (\ell') & (\ell', \ell) \in \text{flow}(S_\ast) \end{cases} \\
& \quad \text{if } \ell = \text{init}(S_\ast) \\
& \quad \text{otherwise}
\end{align*}
\]

\[
\text{RD}_\bullet (\ell) = \left( \text{RD}_\circ (\ell) \backslash \text{kill}_{\text{RD}}(B^\ell) \right) \cup \text{gen}_{\text{RD}}(B^\ell)
\]

where $B^\ell \in \text{blocks}(S_\ast)$
**Very Busy Expressions**

*Kill and gen functions*

\[
\begin{align*}
\text{kill}_{\text{VB}}([x := a]^\ell) &= \{ a' \in \text{AExp}_\bullet | x \in FV(a') \} \\
\text{kill}_{\text{VB}}([\text{skip}]^\ell) &= \emptyset \\
\text{kill}_{\text{VB}}([b]^\ell) &= \emptyset \\
\text{gen}_{\text{VB}}([x := a]^\ell) &= \text{AExp}(a) \\
\text{gen}_{\text{VB}}([\text{skip}]^\ell) &= \emptyset \\
\text{gen}_{\text{VB}}([b]^\ell) &= \text{AExp}(b)
\end{align*}
\]

**Data flow equations:**

\[
\begin{align*}
\text{VB}_{\bullet}^\circ (\ell) &= \begin{cases} \\
\emptyset & \text{if } \ell \in \text{final}(S_\bullet) \\
\bigcap \{ \text{VB}_{\bullet} (\ell') | (\ell', \ell) \in \text{flow}^R(S_\bullet) \} & \text{otherwise}
\end{cases} \\
\text{VB}_{\bullet} (\ell) &= (\text{VB}_{\bullet}^\circ (\ell) \setminus \text{kill}_{\text{VB}}(B^\ell)) \cup \text{gen}_{\text{VB}}(B^\ell) \\
\text{where } B^\ell &\in \text{blocks}(S_\bullet)
\end{align*}
\]
Live Variables

\textit{kill} and \textit{gen} functions

\[ \text{kill}_{LV}(\llbracket x := a \rrbracket^\ell) = \{ x \} \]
\[ \text{kill}_{LV}(\llbracket \text{skip} \rrbracket^\ell) = \emptyset \]
\[ \text{kill}_{LV}(\llbracket b \rrbracket^\ell) = \emptyset \]
\[ \text{gen}_{LV}(\llbracket x := a \rrbracket^\ell) = \text{FV}(a) \]
\[ \text{gen}_{LV}(\llbracket \text{skip} \rrbracket^\ell) = \emptyset \]
\[ \text{gen}_{LV}(\llbracket b \rrbracket^\ell) = \text{FV}(b) \]

data flow equations: \text{LV}$^\text{F}$

\[ \text{LV}_0^\text{F}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in \text{final}(S_\ast) \\ \bigcup \{ \text{LV}_0^\text{F}(\ell') | (\ell', \ell) \in \text{flow}^R(S_\ast) \} & \text{otherwise} \end{cases} \]
\[ \text{LV}_0^\text{F}(\ell) = (\text{LV}_0^\text{F}(\ell) \setminus \text{kill}_{LV}(B^\ell)) \cup \text{gen}_{LV}(B^\ell) \]
where \( B^\ell \in \text{blocks}(S_\ast) \)
Analysis Design
Design of Dataflow Analysis

often the desired results can not be computed directly by a data flow analysis

→Calculate other information and derive the desired results.

Example:

• Dead Assignments → Live Variables
• true dependence → Reaching definitions
Used definition chain

\( \text{UD} : \text{Var} \times \text{Lab} \rightarrow \mathcal{P}(\text{Lab}) \)

\( \text{UD}(x,l) : \) the set of all nodes whose assignments can reach the use of \( x \) at \( l \)

\[ [x:=0]^1; [x:=3]^2; (\text{if } [z=x]^3 \text{ then } [z:=0]^4 \text{ else } [z:=x]^5); [y:=x]^6; [x:=y+z]^7 \]

\[
\text{UD}(x, \ell) = \begin{cases} 
\{ \ell' \mid (x, \ell') \in \text{RD}_{\text{entry}}(\ell) \} & \text{if } x \in \text{gen}_{\text{LV}}(B^\ell) \\
\emptyset & \text{otherwise}
\end{cases}
\]
Monotone Frameworks
The Overall Pattern

Each of the four classical analyses take the form

\[
\text{Analysis}_\circ(\ell) = \begin{cases} 
  \mathcal{I} & \text{if } \ell \in E \\
  \bigcup \{ \text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F \} & \text{otherwise}
\end{cases}
\]

\[
\text{Analysis}_\bullet(\ell) = f_\ell(\text{Analysis}_\circ(\ell))
\]

where

- \( \bigcup \) is \( \cap \) or \( \cup \) (and \( \sqcup \) is \( \cup \) or \( \cap \)),
- \( F \) is either \( \text{flow}(S_\star) \) or \( \text{flow}^R(S_\star) \),
- \( E \) is \( \{ \text{init}(S_\star) \} \) or \( \text{final}(S_\star) \),
- \( \mathcal{I} \) specifies the initial or final analysis information, and
- \( f_\ell \) is the transfer function associated with \( B_\ell \in \text{blocks}(S_\star) \).
Forward versus backward

- The *forward analyses* have $F$ to be $\text{flow}(S_\star)$ and then $\text{Analysis}_\circ$ concerns entry conditions and $\text{Analysis}_\bullet$ concerns exit conditions; the equation system presupposes that $S_\star$ has isolated entries.

- The *backward analyses* have $F$ to be $\text{flow}^R(S_\star)$ and then $\text{Analysis}_\circ$ concerns exit conditions and $\text{Analysis}_\bullet$ concerns entry conditions; the equation system presupposes that $S_\star$ has isolated exits.
Union versus Intersection

- When $\sqcup$ is $\cap$ we require the greatest sets that solve the equations and we are able to detect properties satisfied by *all execution paths* reaching (or leaving) the entry (or exit) of a label; the analysis is called a *must*-analysis.

- When $\sqcap$ is $\sqcup$ we require the smallest sets that solve the equations and we are able to detect properties satisfied by *at least one execution path* to (or from) the entry (or exit) of a label; the analysis is called a *may*-analysis.
Property Spaces

The *property space*, $L$, is used to represent the data flow information, and the *combination operator*, $\cup: \mathcal{P}(L) \to L$, is used to combine information from different paths.

- $L$ is a *complete lattice*, that is, a partially ordered set, $(L, \sqsubseteq)$, such that each subset, $Y$, has a least upper bound, $\bigcup Y$.

- $L$ satisfies the *Ascending Chain Condition*; that is, each ascending chain eventually stabilises (meaning that if $(l_n)_n$ is such that $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \cdots$, then there exists $n$ such that $l_n = l_{n+1} = \cdots$).
Transfer Functions

The set of transfer functions, $\mathcal{F}$, is a set of monotone functions over $L$, meaning that

$$l \subseteq l' \text{ implies } f_\ell(l) \subseteq f_\ell(l')$$

and furthermore they fulfil the following conditions:

- $\mathcal{F}$ contains all the transfer functions $f_\ell : L \rightarrow L$ in question (for $\ell \in \text{Lab}_*$)

- $\mathcal{F}$ contains the identity function

- $\mathcal{F}$ is closed under composition of functions
Frameworks

A *Monotone Framework* consists of:

- a complete lattice, $L$, that satisfies the Ascending Chain Condition; we write $\sqcup$ for the least upper bound operator

- a set $\mathcal{F}$ of monotone functions from $L$ to $L$ that contains the identity function and that is closed under function composition

A *Distributive Framework* is a Monotone Framework where additionally all functions $f$ in $\mathcal{F}$ are required to be distributive:

$$f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$
Instances

An *instance* of a Framework consists of:

- the complete lattice, $L$, of the framework
- the space of functions, $\mathcal{F}$, of the framework
- a finite flow, $F$ (typically $\text{flow}(S_\star)$ or $\text{flow}^R(S_\star)$)
- a finite set of *extremal labels*, $E$ (typically $\{\text{init}(S_\star)\}$ or $\text{final}(S_\star)$)
- an *extremal value*, $\iota \in L$, for the extremal labels
- a mapping, $f_\star$, from the labels $\text{Lab}_\star$ to transfer functions in $\mathcal{F}$
Equations of Instances

$$\text{Analysis}_o(\ell) = \bigsqcup \{ \text{Analysis}_i(\ell') \mid (\ell', \ell) \in F \} \sqcup \iota_E^\ell$$

where $$\iota_E^\ell = \begin{cases} 
\iota & \text{if } \ell \in E \\
\bot & \text{if } \ell \notin E 
\end{cases}$$

$$\text{Analysis}_i(\ell) = f_\ell(\text{Analysis}_o(\ell))$$
## Examples Revisited

<table>
<thead>
<tr>
<th></th>
<th>Available Expressions</th>
<th>Reaching Definitions</th>
<th>Very Busy Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L )</td>
<td>( \mathcal{P}(AExp_\ast) )</td>
<td>( \mathcal{P}(Var_\ast \times Lab_\ast) )</td>
<td>( \mathcal{P}(AExp_\ast) )</td>
<td>( \mathcal{P}(Var_\ast) )</td>
</tr>
<tr>
<td>( \sqsubseteq )</td>
<td>( \sqsubseteq )</td>
<td>( \subseteq )</td>
<td>( \sqsubseteq )</td>
<td>( \subseteq )</td>
</tr>
<tr>
<td>( \sqcup )</td>
<td>( \sqcap )</td>
<td>( \cup )</td>
<td>( \sqcap )</td>
<td>( \cup )</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( AExp_\ast )</td>
<td>( \emptyset )</td>
<td>( AExp_\ast )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \iota )</td>
<td>( \emptyset )</td>
<td>( { (x, ?) \mid x \in FV(S_\ast) } )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( E )</td>
<td>( { \text{init}(S_\ast) } )</td>
<td>( { \text{init}(S_\ast) } )</td>
<td>( \text{final}(S_\ast) )</td>
<td>( \text{final}(S_\ast) )</td>
</tr>
<tr>
<td>( F )</td>
<td>( \text{flow}(S_\ast) )</td>
<td>( \text{flow}(S_\ast) )</td>
<td>( \text{flow}^R(S_\ast) )</td>
<td>( \text{flow}^R(S_\ast) )</td>
</tr>
<tr>
<td>( \mathcal{F} )</td>
<td>( { f : L \to L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g } )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>( f_\ell )</td>
<td>( f_\ell(l) = (l \setminus \text{kill}(B^\ell)) \cup \text{gen}(B^\ell) ) where ( B^\ell \in \text{blocks}(S_\ast) )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td></td>
</tr>
</tbody>
</table>
Bit Vector Frameworks

A *Bit Vector Framework* has

- \( L = \mathcal{P}(D) \) for \( D \) finite
- \( \mathcal{F} = \{ f \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g \} \)

**Examples:**

- Available Expressions
- Live Variables
- Reaching Definitions
- Very Busy Expressions

Bit Vector Frameworks are always Distributive Frameworks
Constant Propagation Frame

An example of a Monotone Framework that is not a Distributive Framework

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

**Example:**

\[
[x := 6]^1; [y := 3]^2; \text{while } [x > y]^3 \text{ do } ([x := x - 1]^4; [z := y * y]^6)
\]

The analysis enables a transformation into

\[
[x := 6]^1; [y := 3]^2; \text{while } [x > 3]^3 \text{ do } ([x := x - 1]^4; [z := 9]^6)
\]
Elements of $L$

\[
\text{State}_{CP} = ((\text{Var}_* \rightarrow Z^\top)_{\perp}, \sqsubseteq)
\]

Idea:

- $\perp$ is the least element: no information is available

- $\widehat{\sigma} \in \text{Var}_* \rightarrow Z^\top$ specifies for each variable whether it is constant:
  - $\widehat{\sigma}(x) \in Z$: $x$ is constant and the value is $\widehat{\sigma}(x)$
  - $\widehat{\sigma}(x) = \top$: $x$ might not be constant
Partial Ordering on $L$

The partial ordering $\subseteq$ on $(\text{Var}_* \rightarrow Z^\top)\bot$ is defined by

$$\forall \hat{\sigma} \in (\text{Var}_* \rightarrow Z^\top)\bot : \bot \subseteq \hat{\sigma}$$

$$\forall \hat{\sigma}_1, \hat{\sigma}_2 \in \text{Var}_* \rightarrow Z^\top : \hat{\sigma}_1 \subseteq \hat{\sigma}_2 \quad \text{iff} \quad \forall x : \hat{\sigma}_1(x) \subseteq \hat{\sigma}_2(x)$$

where $Z^\top = Z \cup \{T\}$ is partially ordered as follows:

$$\forall z \in Z^\top : z \subseteq T$$

$$\forall z_1, z_2 \in Z : (z_1 \subseteq z_2) \Leftrightarrow (z_1 = z_2)$$

\[
\begin{array}{c|c|c|c}
  x \rightarrow 3 & x \rightarrow 4 & x \rightarrow 3 & x \rightarrow 5 \\
  y \rightarrow 5 & y \rightarrow 5 & y \rightarrow 5 & y \rightarrow 4 \\
\end{array}
\]
Flat
Instances

Constant Propagation is a forward analysis, so for the program $S_\star$:

- the flow, $F$, is $\text{flow}(S_\star)$,

- the extremal labels, $E$, is $\{\text{init}(S_\star)\}$,

- the extremal value, $\nu_{CP}$, is $\lambda x. T$, and

- the mapping, $f_{CP}$, of labels to transfer functions is as shown next
Constant Propagation Analysis

\[ \mathcal{A}_{CP} : \text{AExp} \rightarrow (\text{State}_{CP} \rightarrow \mathbb{Z}^\top) \]

\[ \mathcal{A}_{CP}[x]\widehat{\sigma} = \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ \widehat{\sigma}(x) & \text{otherwise} \end{cases} \]

\[ \mathcal{A}_{CP}[n]\widehat{\sigma} = \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ n & \text{otherwise} \end{cases} \]

\[ \mathcal{A}_{CP}[a_1 \ fatop_a \ a_2]\widehat{\sigma} = \mathcal{A}_{CP}[a_1]\widehat{\sigma} \ fatop_a \mathcal{A}_{CP}[a_2]\widehat{\sigma} \]

**transfer functions:** \( f^\ell_{CP} \)

\[ [x := a]^\ell : f^\ell_{CP}(\widehat{\sigma}) = \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ \widehat{\sigma}[x \mapsto \mathcal{A}_{CP}[a]\widehat{\sigma}] & \text{otherwise} \end{cases} \]

\[ [\text{skip}]^\ell : f^\ell_{CP}(\widehat{\sigma}) = \widehat{\sigma} \]

\[ [b]^\ell : f^\ell_{CP}(\widehat{\sigma}) = \widehat{\sigma} \]

Constant Propagation is not a Distributive Framework
Introduction to PAG

- PAG supports the instances of monotone frameworks
- Input: concise specifications
- Output: ANSI C code
- Advantages:
  - rapid implementation
  - integrated debugging facilities
  - short specification
PAG/WWW

• Web interface to PAG
• Restricted features
• Simplified specification language
• Fixed input language

www.program-analysis.com
PAG/WWW vs. PAG

- for WHILE programs only
- restricted specification languages
- restricted syntax
- additionally predefined functionality
- intended for educational purposes
- restricted to a certain iteration algorithm

- full system not bound to a specific language
- additional specification features
- more complicated syntax
- intended to work in a compiler
- also for research purposes
- different interprocedural iteration algorithms
Reaching Definitions Demo

with explanation of textual results
Interpreting Textual Analysis Results

- Automatically labeled program
- Showing Entry (Analysis_o) and Exit (Analysis_...) information
- Procedure parameters are underlined
- \([\text{BOTTOM: } x \rightarrow 7] :\)
  function mapping \(x\) to 7 and everything else to bot (see constant propagation)
Reaching Definitions Demo

with explanation of graphical results
Interpreting Graphical Results

• A picture for each computation step
• Exit information beside outgoing edges
• Entry information not displayed
• Color legend:
  • Red: information is about to be changed
  • Blue: nodes in the worklist
• Node labels: numbered elementary statements
  • $D \rightarrow \bot$
  • $x \rightarrow 7$ is equivalent to $[\rightarrow \text{BOTTOM}: x \rightarrow 7]$
Analysis Specification

[Image of a Romanesco broccoli]
Parts of Specification

- **TYPE**: define the analysis lattice
- **PROBLEM**: define analysis parameters
- **TRANSFER**: define the transfer functions
- **SUPPORT**: define additional functions

Specification in a specialized functional language **FULA** (ML like)
Lattice Specification
Predefined Datatypes

- **snum**  Signed integer
- **bool**  Boolean
- **str**  String
- **Label**  Program label (Lab*)
- **Var**  Program variable (Var*)
- **Proc**  Program procedure
- **Expression**  Non-trivial program expression (Aexp*)
Lattice Construction

- \texttt{set(\langle ld\rangle)}
  - Set over \langle ld\rangle

- \texttt{list(\langle ld\rangle)}
  - List over \langle ld\rangle
  - NOT a lattice!

- \langle ld1\rangle \times \langle ld2\rangle \ldots
  - Tuple space

- \langle ld1\rangle \rightarrow \langle ld2\rangle
  - Function space

- \texttt{flat(<ld>)}
  - Flat lattice

- \texttt{lift(<ld>)}
  - Lifted lattice
Predefined Sets

- LabelSet \( \text{set}(\text{label}) \)
- VarSet \( \text{set}(\text{Var}) \)
- ProcSet \( \text{set}(\text{Proc}) \)
- ExpressionSet \( \text{set}(\text{Expression}) \)
- ExpressionList \( \text{list}(\text{Expression}) \)
Example: Live Variables

• TYPE
  VarSetLifted = lift(VarSet)
Excursion: Solution Algorithm

- In PAG the worklist is initialized only with the start node.
- The programmer must ensure that each node of interest is visited at least once.
  - Add extra ⊥ if the ⊥ of the lattice has a meaningful value.
Lift
Problem Description
Problem Section

- direction  Forward/backward
- carrier     Analysis lattice
- init       Initial value
- init_start Value for the extremal node
- combine    Combination function
Example: Live Variables

- PROBLEM Live_Variabes
  direction : backward
  carrier    : VarSetLifted
  init       : bot
  init_start : lift({})
  combine    : lub
FULA Basics

• Primitives: snum, str, bool
• Sets: { <e1>, ..., <en> } {}
• Lattice elements top bot
• Lists: [ <e1>, ..., <en> ] []
  <e>:list
• Tuples: ( <e1>, ..., <en> )
• Functions: static/dynamic
  f(<e>) [ -> <e> ] f[[<e1>-><e2>]]
FULA Control Structures

- if ... then ... [else ...] endif
- case <e1>, ..., <en> of
  <p1>, ..., <pn> => <e> ...
endcase
- let <p1>=<e1>, ..., <pn>=<en> in <e>
- let <p1> <= <e1>, ... in <e>
  ⇔ if <e1> = top then top else
  if <e1> = bot then bot else
  let <p1> = drop(<e1>) in <e>
endif endif
TRANSFER Section

• Define the exit value in terms of the entry value (@), or vice versa for backward problems

• Definition by cases for the WHILE statements

• Optional matching of the edge type
Example: Live Variables

- **TRANSFER**

```plaintext
let lifeVars <= @ in

IF(exp) =

if @ = top then top
else if @ = bot then bot
else let lifeVars = drop(@) in
  lift(lifeVars lub
      variables(exp))
endif endif
```
Example: Live Variables

- TRANSFER

\[
\text{IF}(\text{exp}) = \\
\text{let } \text{lifeVars} \leq @ \text{ in} \\
\text{lift}(\text{lifeVars lub variables}(\text{exp}))
\]
Example: Live Variables

- TRANSFER
  IF(exp) =

  let lifeVars <= @ in
  lift(lifeVars lub
  variables(exp))
Example: Live Variables

- $\text{ASSIGN}(\text{var, exp}) =$
  
  let lifeVars $\leq \@$ in
  
  lift((lifeVars - var) lub
  
  variables(exp))
FULA Operators

- \( =, \neq \), arithmetic operators
- \(<, \leq, >, \geq\) for lattice and snum
- Boolean \&\&, ||, !
- \((x,y)\#1, x?\{x,y\}\)
- \(\text{lub}, \text{glb}\) for lattices
- \(\text{lift}(x), \text{drop}(x)\)
- \(+\) for string concatenation, set union, set insertion, list append
SUPPORT Section

- Define additional functions
- Each function needs a type declaration
- Definition by cases possible
Example: Live Variables

variables :: Expression -> VarSet
variables(expression) =
{ expVar(exp) | exp in
  subExpressions(expression);
  expType(exp) = "VAR" }