Exploring Interpolants

Philipp Rümmer, Pavle Subotić, Hacked by Srinivas Nedunuri

Uppsala University, Sweden

FMCAD 2013, October 21
1 Background and Motivation

2 Interpolation Abstractions

3 Exploring the space of abstractions

4 Experiments on Software Programs

5 Conclusion
Let \((A \rightarrow \neg B)\) then there exists an interpolant \(I\) for \((A, B)\) such that:

\[ A \rightarrow I \rightarrow \neg B \]

\(I\) refers only to common symbols of \(A, B\)
Introduction

Interpolants in Model Checking

- Interpolants used in model checking to refine abstractions
Introduction

Interpolants in Model Checking

- Interpolants used in model checking to refine abstractions
- Interpolants also used for invariant generation
Introduction

Interpolants in Model Checking
- Interpolants used in model checking to refine abstractions
- Interpolants also used for invariant generation
- For a given interpolation problem several interpolants may exist
Introduction

Interpolants in Model Checking

- Interpolants used in model checking to refine abstractions
- Interpolants also used for invariant generation
- For a given interpolation problem several interpolants may exist
- The choice of interpolants affect if/how a program is verified
Motivation

Analysis using CEGAR

1. Compute an approximation of CFG with respect to a set of predicates
Background and Motivation

Motivation

Analysis using CEGAR

1. Compute an approximation of CFG with respect to a set of predicates
2. Choose a (spurious or genuine) path to error
Motivation

Analysis using CEGAR

1. Compute an approximation of CFG with respect to a set of predicates
2. Choose a (spurious or genuine) path to error
3. If spurious, use interpolation to generate further predicates
Motivation

Motivating Example

```c
i = 0; x = j;    // init
while (i<50) {   // loop
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);  // error location
```

Safety Properties

No feasible path exists that reaches an error state
Motivation

Motivating Example

\begin{verbatim}
i = 0; x = j;  // init
while (i<50) {  // loop
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);  // error location
\end{verbatim}

Counter Example - one loop iteration

\[
\begin{aligned}
\text{init} \\
& i_0 = 0 \land x_0 = j
\end{aligned}
\]
Motivation

Motivating Example

```
i = 0; x = j;  // init
while (i<50) {  // loop
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);  // error location
```

Counter Example - one loop iteration

\[
\begin{align*}
\text{init} & \quad \begin{align*}
i_0 &= 0 \land x_0 = j \\
\end{align*} \\
\text{loop} & \quad \begin{align*}
i_0 &< 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \\
\end{align*}
\end{align*}
\]
Motivation

Motivating Example

```java
i = 0; x = j;  // init
while (i<50) {  // loop
  i++;
  x++;
}
if (j == 0)
  assert (x >= 50);  // error location
```

Counter Example - one loop iteration

```
init
\[
i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \land i_1 \geq 50 \land j = 0 \land x_1 < 50
\]
```

```
loop
```

```
error
```

Motivation

Counter Example - one loop iteration

\[ i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \land i_1 \geq 50 \land j \land x_1 < 50 \]

Interpolation Problem

\[ i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \rightarrow I \]

\[ i_1 \geq 50 \land j \land x_1 < 50 \rightarrow \neg I \]

where \( I \) has symbols only from \( A \) and \( B \)
Motivation

Candidate Interpolant

\[ l_1 = (i_1 \leq 1) \]

The Interpolant

\[
\begin{align*}
A: & \quad i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \rightarrow i_1 \leq 1 \\
B: & \quad i_1 \geq 50 \land j = 0 \land x_1 < 50 \rightarrow \neg i_1 \leq 1 \\
\end{align*}
\]

\[ i_1 \in \text{sym}(A) \text{ and } i_1 \in \text{sym}(B) \]
Motivation

The Problem

- \( i_1 \leq 1 \) eliminates the counter-example
- Results in unrolling the loop - not general enough. May diverge if loop is unbounded
- What we really would like is an inductive invariant
Motivation

A Better Candidate Interpolant

\[ l_2 = (x_1 \geq i_1 + j) \]

The Interpolant

\[
\begin{align*}
A: & \quad i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \rightarrow (x_1 \geq i_1 + j) \\
B: & \quad i_1 \geq 50 \land j = 0 \land x_1 < 50 \rightarrow \neg(x_1 \geq i_1 + j)
\end{align*}
\]

\[ x_1, i_1, j \in \text{sym}(A) \text{ and } x_1, i_1, j \in \text{sym}(B) \]
Motivation

Interpolants

- \((x_1 \geq i_1 + j)\) avoids loop unrolling
- But how do we get \((x_1 \geq i_1 + j)\) instead of \((i_1 \leq 1)\) from the theorem prover?
Interpolants ordered by $\Rightarrow$ (modulo $\equiv$) form a lattice

\[ j \neq 0 \lor i_1 \leq 49 \lor x_1 \geq 50 \]

\[ i_1 \leq 49 \]
\[ i_1 \leq 2 \]
\[ i_1 \leq 1 \]
\[ i_1 = 1 \]

\[ j \neq 0 \lor x_1 \geq i_1 \]
\[ x_1 \geq i_1 + j \]
\[ x_1 = i_1 + j \]

\[ x_1 = j + 1 \land i_1 = 1 \]

$\top$
$\bot$

Rümmer, Subotić
Exploring Interpolants

FMCA 2013, October 21

15 / 42
Interpolants ordered by \( \Rightarrow \) (modulo \( \equiv \)) form a lattice

\[
\begin{align*}
&j \neq 0 \lor i_1 \leq 49 \lor x_1 \geq 50 \quad I_T \\
i_1 \leq 49 &\quad \vdots \\
i_1 \leq 2 &\quad j \neq 0 \lor x_1 \geq i_1 \\
i_1 \leq 1 &\quad l_2 \\
i_1 = 1 \\
&x_1 = j + 1 \land i_1 = 1 \quad l_\perp
\end{align*}
\]

How to guide the construction of the lattice elements (interpolants)?

- Key is interpolation abstractions

How to find ”good” interpolant?
Quick background on Lattices

Definition (Poset)
A poset is a set $S$ equipped with a partial ordering $\sqsubseteq$. A poset $\langle S, \sqsubseteq \rangle$ is bounded if it has a least element $\bot$ and a greatest element $\top$.

Definition (Lattice, Complete Lattice, Sub-lattice)
A lattice $L$ is a poset $\langle S, \sqsubseteq \rangle$ such that $\sqcup(a, b)$ and $\sqcap(a, b)$ exist for all $a, b \in S$. $L$ is a complete lattice if all non-empty subsets of $S$ have a greatest lower bound and least upper bound. A complete lattice is bounded by definition. A non-empty subset $X$ of $S$ forms a sub-lattice if $\sqcup(a, b) \in X$ and $\sqcap(a, b) \in X$ for all $a, b \in X$. 
Outline of Approach

Pre-process the interpolation query
Outline of Approach

Pre-process the *interpolation query*

- General, prover independent framework
Outline of Approach

Pre-process the interpolation query

- General, prover independent framework
- Generate several interpolants for a given interpolation problem
Outline of Approach

Pre-process the *interpolation query*

- General, prover independent framework
- Generate several interpolants for a given interpolation problem
- Incorporate programmer knowledge in defining interpolant quality
1. Background and Motivation

2. Interpolation Abstractions

3. Exploring the space of abstractions

4. Experiments on Software Programs

5. Conclusion
Abstractions in the Example

Verification Condition

\[
\begin{align*}
A & : i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \land i_1 \geq 50 \\
B & : j = 0 \land x_1 < 50
\end{align*}
\]

- Step 1: Rename common variables in \( A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B] \)

In the example: common symbols are \( \{ j, i_1, x_1 \} \)

\[
\begin{align*}
A[\bar{s}_A, \bar{s}'] & = i_0 = 0 \land x_0 = j' \land i_0 < 50 \land i'_1 = i_0 \land x'_1 = x_0 \\
B[\bar{s}'', \bar{s}_B] & = i''_1 \geq 50 \land j'' = 0 \land x''_1 < 50
\end{align*}
\]
Abstractions in the Example

- Step 1: Rename common symbols in $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$
- Step 2: Add ”templates”\(^a\) capturing limited knowledge

\(^a\)I prefer the term ”template parameter” to ”template”

In the example: template parameters are $\{j, x_1 - i_1\}$

$$A[\bar{s}_A, \bar{s}]^\# = i_0 = 0 \land x_0 = j' \land i_0 < 50 \land i_1' = i_0 \land x_1' = x_0 \land x_1' - i_1' = x_1 - i_1 \land j'$$

$$B[\bar{s}, \bar{s}_B]^\# = i_1'' \geq 50 \land j'' = 0 \land x_1'' < 50 \land x_1 - i_1 = x_1'' - i_1'' \land j = j''$$

\[R_A[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} t_i[\bar{s}'] = t_i[\bar{s}], \quad R_B[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} t_i[\bar{s}] = t_i[\bar{s}'']\]
Example

Interpolation Problem $A \land B$
Example

With abstraction generated by template parameters $x - y$
Example

Blocks Interpolants $x \geq 4$ etc.
Example

Allows interpolants $x \geq y$ etc.
Definitions

Definition (Abstraction)

An **interpolation abstraction** is a pair \((R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])\) of formulae with the property that \(R_A[\bar{s}, \bar{s}]\) and \(R_B[\bar{s}, \bar{s}]\) are valid i.e., \(\text{Id}[\bar{s}', \bar{s}] \Rightarrow R_A[\bar{s}', \bar{s}]\) and \(\text{Id}[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']\).

Definition (Abstract Interpolation Problem)

- \(A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]\) is the **concrete interpolation problem**.
- \((A[\bar{s}_A, \bar{s}'] \land R_A[\bar{s}, \bar{s}']) \land (R_B[\bar{s}'', \bar{s}] \land B[\bar{s}'', \bar{s}_B])\) is called the **abstract interpolation problem**;

Definition (Feasible Abstractions)

Assuming that the concrete interpolation problem is solvable, we call an interpolation abstraction **feasible** if also the abstract interpolation problem is solvable, and **infeasible** otherwise.
Natural classes of Abstractions

- **Term interpolation abstractions**, constructed from a set of terms \( \{ t_1, t_2, \ldots, t_n \} \)

\[
R^T_A[\vec{s}', \vec{s}] = \bigwedge_{i=1}^{n} t_i[\vec{s}'] = t_i[\vec{s}], \quad R^T_B[\vec{s}, \vec{s}''] = \bigwedge_{i=1}^{n} t_i[\vec{s}] = t_i[\vec{s}'']
\]

(same possible for inequalities)

- **Predicate interpolation abstractions**, constructed from \( \{ \phi_1, \phi_2, \ldots, \phi_n \} \)

\[
R^{Pred}_A[\vec{s}', \vec{s}] = \bigwedge_{i=1}^{n} (\phi_i[\vec{s}'] \rightarrow \phi_i[\vec{s}]), \quad R^{Pred}_B[\vec{s}, \vec{s}''] = \bigwedge_{i=1}^{n} (\phi_i[\vec{s}] \rightarrow \phi_i[\vec{s}''])
\]

- Quantified interpolation abstractions (supply terms with free variables)

- …
Soundness and Completeness

Lemma (Soundness)

Every interpolant of the abstract interpolation problem is also an interpolant of the concrete interpolation problem (but in general not vice versa).

Lemma (Completeness)

Suppose \( A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B] \) is an interpolation problem with interpolant \( I[\bar{s}] \), such that both \( A[\bar{s}_A, \bar{s}] \) and \( B[\bar{s}, \bar{s}_B] \) are satisfiable. Then there is a feasible interpolation abstraction such that every abstract interpolant is equivalent to \( I[\bar{s}] \).
1 Background and Motivation

2 Interpolation Abstractions

3 Exploring the space of abstractions

4 Experiments on Software Programs

5 Conclusion
Exploring the space of abstractions

Exploring Abstractions

- How do we find good interpolation abstractions?
- Can be done in two steps:
  - Define a base vocabulary of “interesting” templates (building blocks for interpolants)
  - Search for **maximum feasible** interpolation abstractions in this language
Exploring Abstractions

How do we find good interpolation abstractions?
Can be done in two steps:
- Define a base vocabulary of “interesting” templates (building blocks for interpolants)
- Search for maximum feasible interpolation abstractions in this language

Definition (Abstraction lattice)
Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$. An abstraction lattice is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a monotonic mapping $\mu$ from elements of $\langle L, \sqsubseteq_L \rangle$ to interpolation abstractions $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ with the property that $\mu(\bot) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$. 

Rümmer, Subotić
Exploring Interpolants
FMCAD 2013, October 21
Abstraction lattice template base set \( \{x_1 - i_1, i_1, j\} \)
Sub-lattices of interpolant lattice

\[ j \neq 0 \lor i_1 \leq 49 \lor x_1 \geq 50 \]

\[ i_1 \leq 49 \]

\[ i_1 \leq 2 \]

\[ i_1 \leq 1 \]

\[ i_1 = 1 \]

\[ x_1 = j + 1 \land i_1 = 1 \]

\[ x_1 \geq i_1 + j \]

\[ j \neq 0 \lor x_1 \geq i_1 \]

\[ l_2 \]

\[ l_1 \]

\[ l_{\perp} \]
Overall Architecture

```
Verifier

Interpolation Engine

CEGAR loop

predicates

Query
```
Overall Architecture

- Light-weight static analysis
- Verifier
- Interpolation Engine
- Exploration of Abstraction Lattice
  - Domain knowledge
  - Template lattice
  - Counter example
  - Feasible abstractions
- Interpolation Abstraction
  - Predicates
  - Abstract query(s)
  - CEGAR loop
1 Background and Motivation

2 Interpolation Abstractions

3 Exploring the space of abstractions

4 Experiments on Software Programs

5 Conclusion
Experiments

Experiment Setup

- Extended the Eldarica model checker with our approach
- Experiments on Horn clause benchmarks generated from programs
- Pre-computed templates of the form \( \{x, y, x - y, x + y\} \)
  Typically 15–300 templates
- Costs assigned to templates to define preference
## Experiments

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Eldarica</th>
<th>Eldarica-ABS</th>
<th>Flata</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>sec</td>
<td>N</td>
<td>sec</td>
</tr>
<tr>
<td><strong>C programs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boustrophedon (C)</td>
<td>*</td>
<td>*</td>
<td>10</td>
<td>10.7</td>
</tr>
<tr>
<td>boustrophedon_expanded (C)</td>
<td>*</td>
<td>*</td>
<td>11</td>
<td>7.7</td>
</tr>
<tr>
<td>halbwachs (C)</td>
<td>*</td>
<td>*</td>
<td>53</td>
<td>2.4</td>
</tr>
<tr>
<td>gopan (C)</td>
<td>17</td>
<td>22.2</td>
<td>62</td>
<td>57.0</td>
</tr>
<tr>
<td>rate_limiter (C)</td>
<td>11</td>
<td>2.7</td>
<td>11</td>
<td>19.1</td>
</tr>
<tr>
<td>anubhav (C)</td>
<td>1</td>
<td>1.7</td>
<td>1</td>
<td>1.6</td>
</tr>
<tr>
<td>cousot (C)</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>7.7</td>
</tr>
<tr>
<td>bubblesort (E)</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>insdel (C)</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>insertsort (E)</td>
<td>1</td>
<td>1.8</td>
<td>1</td>
<td>1.7</td>
</tr>
<tr>
<td>listcounter (C)</td>
<td>*</td>
<td>*</td>
<td>8</td>
<td>2.0</td>
</tr>
<tr>
<td>listcounter (E)</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>listreversal (C)</td>
<td>1</td>
<td>1.9</td>
<td>1</td>
<td>1.9</td>
</tr>
<tr>
<td>mergesort (E)</td>
<td>1</td>
<td>2.9</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>selectionsort (E)</td>
<td>1</td>
<td>2.4</td>
<td>1</td>
<td>2.4</td>
</tr>
<tr>
<td>rotation vc.1 (C)</td>
<td>7</td>
<td>2.0</td>
<td>7</td>
<td>0.3</td>
</tr>
<tr>
<td>rotation vc.2 (C)</td>
<td>8</td>
<td>2.7</td>
<td>8</td>
<td>0.2</td>
</tr>
<tr>
<td>rotation vc.3 (C)</td>
<td>0</td>
<td>2.3</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>rotation.1 (E)</td>
<td>3</td>
<td>1.8</td>
<td>3</td>
<td>1.8</td>
</tr>
<tr>
<td>split vc.1 (C)</td>
<td>18</td>
<td>3.9</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>split vc.2 (C)</td>
<td>*</td>
<td>*</td>
<td>18</td>
<td>1.1</td>
</tr>
<tr>
<td>split vc.3 (C)</td>
<td>0</td>
<td>2.8</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>Recursive Horn SMT-LIB Benchmarks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addition (C)</td>
<td>1</td>
<td>0.7</td>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>bfprt (C)</td>
<td>*</td>
<td>*</td>
<td>5</td>
<td>8.3</td>
</tr>
<tr>
<td>binarysearch (C)</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>buildheap (C)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>countZero (C)</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>disjunctive (C)</td>
<td>10</td>
<td>2.4</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>floodfill (C)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>gcd (C)</td>
<td>4</td>
<td>1.2</td>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>identity (C)</td>
<td>2</td>
<td>1.1</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>merge-leq (C)</td>
<td>3</td>
<td>1.1</td>
<td>7</td>
<td>7.0</td>
</tr>
</tbody>
</table>
1 Background and Motivation

2 Interpolation Abstractions

3 Exploring the space of abstractions

4 Experiments on Software Programs

5 Conclusion
Summary

A semantic, solver-independent framework for guiding interpolant search
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
Conclusion

Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation
- General framework
A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
  - Each query can have a specific lattice, lattices can be infinite etc.
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
  - Each query can have a specific lattice, lattices can be infinite etc.
  - Applicable to various logics, not restricted to arithmetic
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
  - Each query can have a specific lattice, lattices can be infinite etc.
  - Applicable to various logics, not restricted to arithmetic

- Templates, but interpolants still constructed by theorem prover
  ⇒ Arbitrary Boolean structure, etc., allowed
Some Related Work

- Syntactic restrictions (R. Jhala and K. L. McMillan, TACAS 06)
- Interpolant strength (V. D'Silva VMCAI 10)
- Beautiful Interpolants (A. Albarghouthi, K. L. McMillan, CAV 13)
- Term abstraction (F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, LPAR 12)
Thank you - Questions
Finding Abstractions

**Algorithm 1:** Exploration algorithm

**Input:** Interpolation problem $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$

**Result:** Set of maximal feasible interpolation abstractions

1. if $\bot$ is infeasible then
2.  return $\emptyset$;
3. end

4. $\text{Frontier} \leftarrow \{\text{maximise}(\bot)\}$;
5. while $\exists$ feasible $\text{elem} \in L$, incomparable with $\text{Frontier}$ do
6.  $\text{Frontier} \leftarrow \text{Frontier} \cup \{\text{maximise}(\text{elem})\}$;
7. end

8. return $\text{Frontier}$;
Finding Abstractions

**Algorithm 2: Maximisation algorithm**

**Input:** Feasible element: \( elem \)

**Result:** Maximal feasible element

1. while \( \exists \) feasible successor \( fs \) of \( elem \) do
2.   pick element \( middle \) such that \( fs \sqsubseteq_L middle \sqsubseteq_L \top \);
3.   if \( middle \) is feasible then
4.     \( elem \leftarrow middle \);
5.   else
6.     \( elem \leftarrow fs \);
7.   end
8. end
9. return \( elem \);