Exploring Interpolants

Philipp Rümmer, **Pavle Subotić**

Uppsala University, Sweden

COST Meeting, October 17

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
  - Incorporates *domain specific knowledge*

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
  - Incorporates *domain specific knowledge*
  - *Semantic* in nature

Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
  - Incorporates *domain specific knowledge*
  - *Semantic* in nature
  - Prover independent

Preliminaries

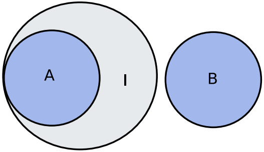
Craig Interpolants

Let $(A \wedge B = \mathit{false})$ then there exists an interpolant I for (A, B) such that:

$$A \rightarrow I$$

$$B \rightarrow \neg I$$

I refers only to common symbols of A, B



Motivation

Motivating Example

```
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

Analysis using CEGAR

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

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

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

Counter Example - one loop iteration

$$\overbrace{i_0 = 0 \wedge x_0 = j}^{\text{init}}$$

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

$$\overbrace{i_0 = 0 \land x_0 = j}^{\text{init}} \land \overbrace{i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1}^{\text{loop}}$$

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

$$\overbrace{i_0 = 0 \wedge x_0 = j}^{\text{init}} \wedge \overbrace{i_0 < 50 \wedge i_1 = i_0 + 1 \wedge x_1 = x_0 + 1}^{\text{loop}} \wedge \overbrace{i_1 \geq 50 \wedge j = 0 \wedge x_1 < 50}^{\text{error}}$$

Motivation

Counter Example - one loop iteration

$$\underbrace{i_0 = 0 \wedge x_0 = j \wedge i_0 < 50 \wedge i_1 = i_0 + 1 \wedge x_1 = x_0 + 1}_{A} \wedge \underbrace{i_1 \geq 50 \wedge j = 0 \wedge x_1 < 50}_{B}$$

Interpolation Problem

$$\underbrace{i_0 = 0 \wedge x_0 = j \wedge i_0 < 50 \wedge i_1 = i_0 + 1 \wedge x_1 = x_0 + 1}_{A} \rightarrow I$$

$$\underbrace{i_1 \geq 50 \wedge j = 0 \wedge x_1 < 50}_{B} \rightarrow \neg I$$

where I has symbols only from A and B

Motivation

Candidate Interpolant

$$I_1 = (i_1 \leq 1)$$

The Interpolant

$$\underbrace{i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1}_{A} \rightarrow i_1 \leq 1 \checkmark$$

$$\underbrace{i_1 \geq 50 \land j = 0 \land x_1 < 50}_{B} \rightarrow \neg i_1 \leq 1 \checkmark$$

$i_1 \in sym(A)$ and $i_1 \in sym(B)$ $\checkmark$

Motivation

The Problem

- $(i_1 \leq 1)$ eliminates the counter-example
- Results in unrolling the loop - not *general* enough
- What we really would like is an **inductive invariant**

Motivation

A Better Candidate Interpolant

$$I_2 = (x_1 \geq i_1 + j)$$

The Interpolant

$$\underbrace{i_0 = 0 \wedge x_0 = j \wedge i_0 < 50 \wedge i_1 = i_0 + 1 \wedge x_1 = x_0 + 1}_{A} \rightarrow (x_1 \geq i_1 + j)\checkmark$$

$$\underbrace{i_1 \geq 50 \wedge j = 0 \wedge x_1 < 50}_{B} \rightarrow \neg(x_1 \geq i_1 + j)\checkmark$$

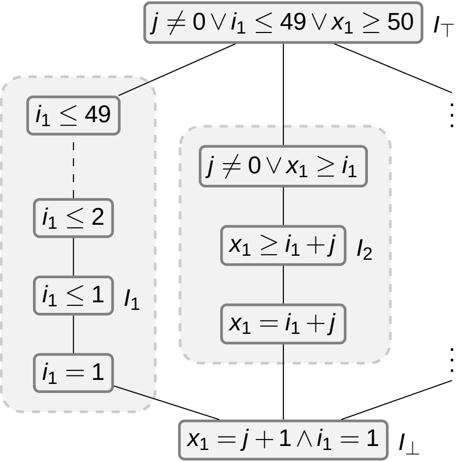
$x_1, i_1, j \in sym(A)$ and $x_1, i_1, j \in sym(B)$ $\checkmark$

Motivation

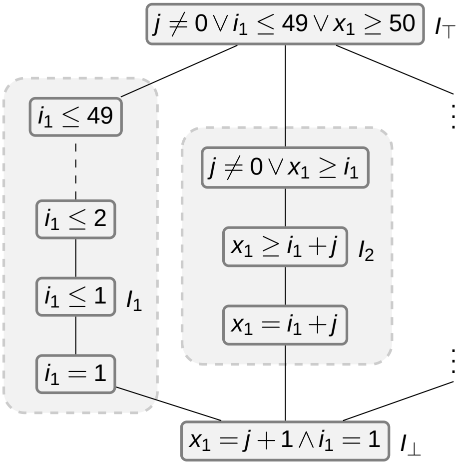
Interplants

- $(x_1 \geq i_1 + j)$ avoids loop unrolling
- But how do we get $(x_1 \geq i_1 + j)$ instead of $(i_1 \leq 1)$ from the theorem prover?

Interpolant lattice for the example



Interpolant lattice for the example



- How to navigate in lattice?
- How to compare "quality" of interpolants?

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)

Our Approach

Pre-process the *interpolation query*

Our Approach

Pre-process the *interpolation query*

- General, prover independent framework

Our Approach

Pre-process the *interpolation query*

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

Our Approach

Pre-process the *interpolation query*

- General, prover independent framework
- Generate several interpolants for a given interpolation problem
- Incorporate domain specific knowledge in defining interpolant quality

Outline

Abstractions in the Example

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

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

$$
\begin{aligned}
A[\bar{s}_A, \bar{s}'] &= i_0 = 0 \wedge x_0 = j' \wedge i_0 < 50 \wedge i_1' = i_0 \wedge x_1' = x_0 \\
B[\bar{s}'', \bar{s}_B] &= i_1'' \geq 50 \wedge j'' = 0 \wedge x_1'' < 50
\end{aligned}
$$

Abstractions in the Example

- Step 1: Rename common symbols in $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$
- Step 2: Add templates capturing limited knowledge

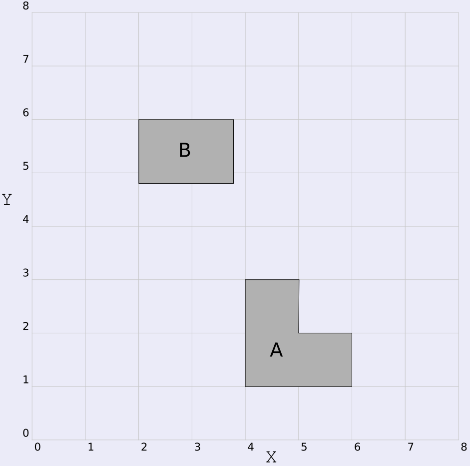
In the example: templates are $\{j, x_1 - i_1\}$

$$A[\bar{s}_A, \bar{s}]^\sharp \;=\; i_0 = 0 \wedge x_0 = j' \wedge i_0 < 50 \wedge i_1' = i_0 \wedge x_1' = x_0 \;\wedge\; \underbrace{x_1' - i_1' = x_1 - i_1 \wedge j' = j}_{R_A[\bar{s}', \bar{s}]}$$

$$B[\bar{s}, \bar{s}_B]^\sharp \;=\; i_1'' \geq 50 \wedge j'' = 0 \wedge x_1'' < 50 \;\wedge\; \underbrace{x_1 - i_1 = x_1'' - i_1'' \wedge j = j''}_{R_B[\bar{s}, \bar{s}'']}$$

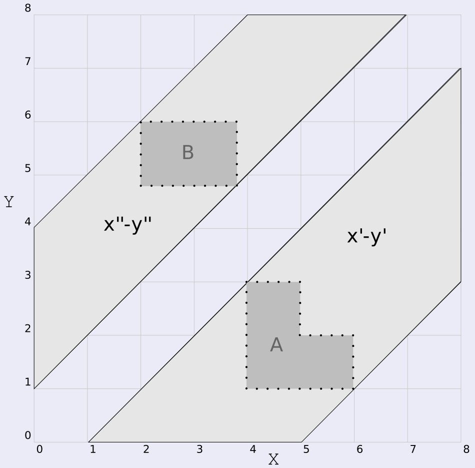
Example

Interpolation Problem $A \wedge B$



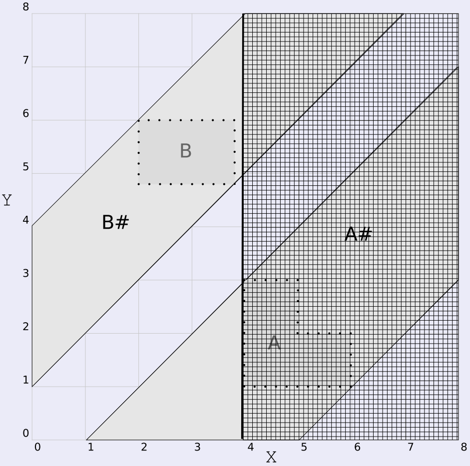
Example

With abstraction generated by template $x - y$



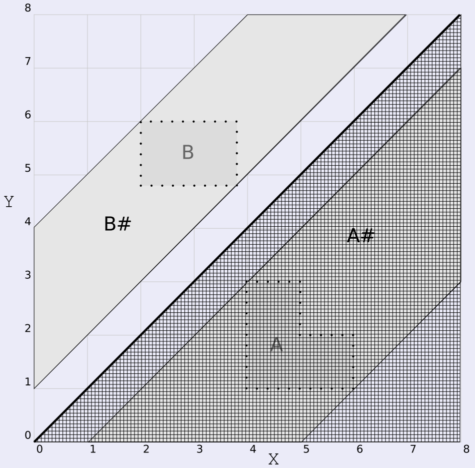
Example

Blocks Interpolants $x \geq 4$ etc.

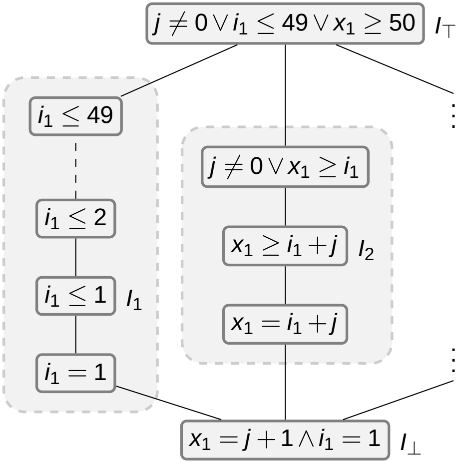


Example

Allows interpolants $x \geq y$ etc.



Interpolant sub-lattice for templates $\{i_1\}$ and $\{j, x_1 - i_1\}$



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., $Id[\bar{s}', \bar{s}] \Rightarrow R_A[\bar{s}', \bar{s}]$ and $Id[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']$.

Definition (Abstract Interpolation Problem)

- $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is the **concrete interpolation problem**.
- $\big(A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}, \bar{s}']\big) \wedge \big(R_B[\bar{s}'', \bar{s}] \wedge B[\bar{s}'', \bar{s}_B]\big)$ is called **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 $\{t_1, t_2, \ldots, t_n\}$

$$R_A^T[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} t_i[\bar{s}'] = t_i[\bar{s}], \quad R_B^T[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} t_i[\bar{s}] = t_i[\bar{s}'']$$

- (same possible for inequalities)
- **Predicate interpolation abstractions**, constructed from $\{\phi_1, \phi_2, \ldots, \phi_n\}$

$$R_A^{Pred}[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} \left(\phi_i[\bar{s}'] \to \phi_i[\bar{s}]\right), \quad R_B^{Pred}[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} \left(\phi_i[\bar{s}] \to \phi_i[\bar{s}'']\right)$$

- Quantified interpolation abstractions
- $\cdots$

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 $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ *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* $I[\bar{s}]$*.*

Exploring Interpolants

- 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

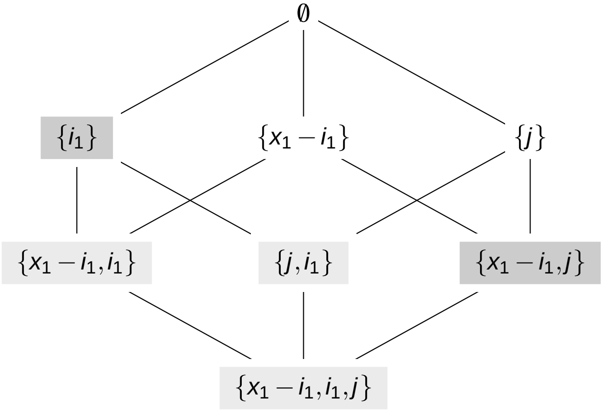
Exploring Interpolants

- 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

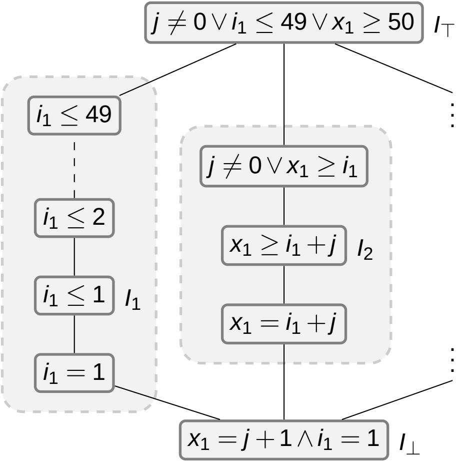
Definition (Abstraction lattice)

Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$. 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 μ 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 $\mu(\bot) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$.

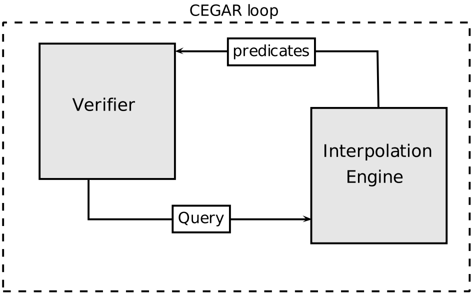
Abstraction lattice template base set $\{x_1 - i_1, i_1, j\}$



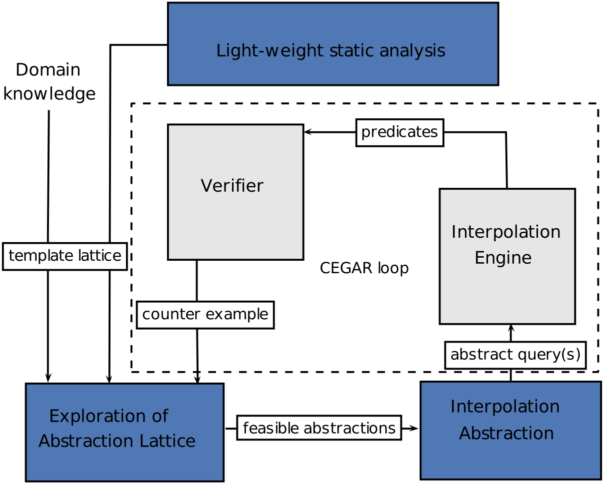
Sub-lattices of interpolant lattice



Overall Architecture



Overall Architecture



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

Benchmark	Eldarica		Eldarica-ABS		Flata	Z3
	N	sec	N	sec	sec	sec
C programs						
boustrophedon (C)	*	*	10	10.7	*	0.1
boustrophedon_expansed (C)	*	*	11	7.7	*	0.1
halbwachs (C)	*	*	53	2.4	*	0.1
gopan (C)	17	22.2	62	57.0	0.4	349.5
rate_limiter (C)	11	2.7	11	19.1	1.0	0.1
anubhav (C)	1	1.7	1	1.6	0.9	*
cousot (C)	*	*	3	7.7	0.7	*
bubblesort (E)	1	2.8	1	2.3	77.6	0.3
insdel (C)	1	0.9	1	0.9	0.7	0.0
insertsort (E)	1	1.8	1	1.7	1.3	0.1
listcounter (C)	*	*	8	2.0	0.2	*
listcounter (E)	1	0.9	1	0.9	0.2	0.0
listreversal (C)	1	1.9	1	1.9	4.9	*
mergesort (E)	1	2.9	1	2.6	1.1	0.2
selectionsort (E)	1	2.4	1	2.4	1.2	0.2
rotation_vc.1 (C)	7	2.0	7	0.3	1.9	0.2
rotation_vc.2 (C)	8	2.7	8	0.2	2.2	0.3
rotation_vc.3 (C)	0	2.3	0	0.2	2.3	0.0
rotation.1 (E)	3	1.8	3	1.8	0.5	0.1
split_vc.1 (C)	18	3.9	17	3.2	*	1.1
split_vc.2 (C)	*	*	18	1.1	*	0.2
split_vc.3 (C)	0	2.8	0	1.5	*	0.0
Recursive Horn SMT-LIB Benchmarks						
addition (C)	1	0.7	1	0.8	0.4	0.0
bfprt (C)	*	*	5	8.3	-	0.0
binarysearch (C)	1	0.9	1	0.9	-	0.0
buildheap (C)	*	*	*	*	-	*
countZero (C)	2	2.0	2	2.0	-	0.0
disjunctive (C)	10	2.4	5	5.0	0.2	0.3
floodfill (C)	*	*	*	*	41.2	0.1
gcd (C)	4	1.2	4	2.0	-	*
identity (C)	2	1.1	2	2.1	-	0.1
merge-leq (C)	3	1.1	7	7.0	15.7	0.1

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)

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

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

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
  $\Rightarrow$ Arbitrary Boolean structure, etc., allowed

Summary

Applications (ongoing work)

- Software programs with heap, other datatypes
- Timed systems
- Reachability in Petri nets/Vector addition systems

Thank you - Questions

Finding Abstractions

Algorithm 1: Exploration algorithm

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

Result: Set of maximal feasible interpolation abstractions

```
if ⊥ is infeasible then
    return ∅;
end
Frontier ← {maximise(⊥)};
while ∃ feasible elem ∈ L, incomparable with Frontier do
    Frontier ← Frontier ∪ {maximise(elem)};
end
return Frontier;
```

Finding Abstractions

Algorithm 2: Maximisation algorithm

Input: Feasible element: *elem*

Result: Maximal feasible element

```
while ∃ feasible successor fs of elem do
    pick element middle such that fs ⊑_L middle ⊑_L ⊤;
    if middle is feasible then
        elem ← middle;
    else
        elem ← fs;
    end
end
return elem;
```