Equality Specifications section, Algorithms topic, Lecture 7

Pavel Panchekha
CS 6110, U of Utah
28 January 2020
The Approach

First-order statement
\( \forall \exists p \)

Standard Form \( p' \)

Domain solvers
\( q_1 \land q_2 \land q_3 \)

Proofs of
\( q_1, q_2, q_3 \)

Proof of \( \forall \exists p \)

or

Proof of \( \neg \forall \exists p \)
DPLL with Theories

Want **extension to first-order theories**

**Statement** $p$

$$((\neg a_1 \lor a_2) \land (\neg b_1 \lor b_2) \land \ldots)$$

**Assignment** $\Gamma$

$$a_1 \land \neg b_2 \land \ldots$$

The $a_i, b_i, \ldots$ are **literals from the theory**

$$((\neg (x < 0) \lor (x \times x > 0)) \land (\neg (x = 0) \lor (x \times x = 0)) \land \ldots)$$

$$x < 0 \land (x \times x = 0) \land \ldots$$
# Standard Form

## Notation

### Negation

\[ \neg G \]

### Implication

\[ \rightarrow \]

### Equality

\[ = \]

### AND

\[ \land \]

### OR

\[ \lor \]

### MORE

\[ \geq \]

### LESS

\[ \leq \]

### NOT

\[ \neg \]

### Neither

\[ ? \]

### Choice

\[ \triangleright \]

### SAT

\[ \text{SAT} \]

### Choose

\[ \text{Choose C = D} \]

### Theories disagree

\[ \text{C = D? Theories disagree} \]

## Table

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
<th>( D )</th>
<th>( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Class Progress

Logical reasoning

Program logics

Static analysis

First-order Logic

Decision Procedures

Mixing Theories

Equality

Integers

Arrays
Equality

Turning syntactic equality into **pointer equality**
- Pointer sharing in tree structures

**Solving** equality queries with congruence closure
- Rebuilding to ensure accuracy

**Model-based** quantifier instantiation
- And why the heuristic doesn’t work in general
Term Databases

A model of the reflection axioms
The Solver Query

Statement $p$
$(\neg a_1 \lor a_2) \land (\neg b_1 \lor b_2) \land \ldots$

Assignment $\Gamma$
$a_1 \land \neg b_2 \land \ldots$

Today: a solver for equality queries

$\Gamma = (s_1 = t_1) \land (s_2 = t_2) \land (s_3 \neq t_3) \land \ldots$

$\Gamma = a = b \land b = c \land a \neq c$

$\Gamma = a = b \land f(a) \neq f(b)$

$\Gamma = f(a) = b \land f(b) = a \land a \neq b$
Modeling

Goal is to check all relevant instances of **axioms**

<table>
<thead>
<tr>
<th>Reflection</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall x, x = x$</td>
<td>$\forall x, \forall y, x = y \rightarrow f(x) = f(y)$</td>
</tr>
</tbody>
</table>

Intuitive approach **starts with the true** equalities

Build **model** of assumptions, check conclusions
Term Representation

Query is a list of pairs of terms, plus a flag for negation

\[ \Gamma = (s_1 = t_1) \land (s_2 = t_2) \land (s_3 \neq t_3) \land \ldots \]

Type `EqQuery` = List (Boolean, Term, Term)

Terms are syntax trees:
Subterm Sharing
Avoid **multiple copies** of one expression

**Make equal things identical** to enforce equality axioms
Term Hashing

Create **term database** by assigning names to terms

```python
class TermDB:
    def add(this : TermDB, term : Term) → Int

    def add(this, term):
        arg_names = map(this.add, term.arguments)
        key = (term.function, arg_names)
        if key not in this.names:
            this.names[key] = len(this.names)
        return this.names[key]
```

add recursively **reuses names** when possible
Term Database

Equality solver uses term database

```python
def equality_solver(query : EqQuery) → Bool:
    db = TermDb()
```

Term database models reflection axiom

Same expression has identical name in database

Need ability to add new equalities
Example

How many terms in the term database?

\[ f(g(a,1),2) = 5 \land g(f(1,5), g(a,1)) = 3 \land g(a,1) \neq a \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>g</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Checking

Need to **check satisfiability** after adding all terms

\[
\Gamma = s_1 = t_1 \land s_2 = t_2 \land s_3 \neq t_3 \land s_4 \neq t_4
\]

\[
\neg \Gamma = (s_1 = t_1 \land s_2 = t_2) \rightarrow (s_3 = t_3 \lor s_4 = t_4)
\]

**Check disequalities** via names

```python
neqs = [(s, t) for (iseq, s, t) in query if not iseq]

return not any(db.add(s) == db.add(t) for (s, t) in neqs)
```
Equivalence Graphs

Adding assumptions to the term database
Adding equalities

\[ \Gamma = f(a) = b \land f(b) = a \land a \neq b \]

Need to add equalities to the term database

Must respect substitution rule:

\[ a = b \rightarrow f(a) = f(b) \]
Equivalence Classes

\[ f(a) \quad \text{and} \quad f(b) \]
Equivalence Classes

\[ f(b) = a \]
Equivalence Classes

\[ f(a) = b \quad f(b) = a \]
Equivalence Classes

$\begin{align*}
&f(a) = b \quad f(b) = a
\end{align*}$

How to represent equivalence classes in code?
Equivalence Classes

How to represent equivalence classes in code?

class TermDB:
    eqclass : Dict Int Int
        Name Class

Initially, each node in its own class:

def add(this, term):
    # ...
    this.eqclass[name] = name
Equivalence Classes

On merging, **pick one class** as the winner

```python
def merge(term1, term2):
    class1 = this.classof(this.add(term1))
    class2 = this.classof(this.add(term2))
    this.eqclass[class1] = class2
```

Class of a node determined **recursively**:

```python
def classof(name):
    cls = this.eqclass[name]
    if cls == name: return pls
    else: return this.classof(cls)
```
Congruence Closure

\[ a = b \]
Congruence Closure

Merging two nodes can make other nodes equal

Need to merge those nodes to ensure correctness
for (f, args), name in this.names.items():
    arg_classes = map(this.classof, args)
    name2 = this.add(f, arg_classes)
    this.eqclass[name] = name2
Checking

Use "merge" on true equalities:

```python
eqs = [(s, t) for (iseq, s, t) in query if iseq]

for s, t in eqs:
    this.merge(s, t)
    this.merge_upward()
```

Check false equalities at the end
Example

How many classes in the term database?

\[ f(a, b) = c \land f(b, c) = f(c, b) \land \]
\[ f(f(a, b), b) = f(a, a) \land f(a, b) = f(a, a) \]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>f</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Speed

Lots of ways to **make this faster**

- Fix up congruence closure **less often** (once at the end?)
- Pick which **class name to keep** when you merge
- **Cache** “classof” lookup so that it’s fast
- **Lay out data structures** to be parallel, cache-friendly

Core data structure is as described
Course Updates

Assignment 2
Assignment 2

Assignment 2 due on Thursday

Submit early so you’re not late
Check submission for any mistakes
Please put your **name in file name**; it helps grading

Lots of good questions on Piazza
- How to encode KenKen into SAT
- How fast to expect the SMT solution to be
- Hints on how the bonus problem should work
Quantifier Heuristics

Where quantifiers can and can’t be useful
Quantified Equalities

Quantifiers convenient to **express arithmetic laws**:

\[ \forall x \forall y, x + y = y + x \]
\[ \forall x \forall y, (x + y) + z = x + (y + z) \]
\[ \forall x \forall y, (x + y) - y = x \]

Can **equivalence graphs use quantified** equalities?
Instantiation

Quantified equality is **infinitely-many** equalities

But only some of them are **relevant**

Instantiate based on **content of e-graph** (model-based)

\[ \forall x \forall y, x + y = y + x \]
E-matching

Goal: find places a quantified equality matches

1. Loop over all e-classes
2. Find terms with matching function name
3. Recurse on arguments
4. Check variables bound identically
   \[ \forall x, x - x = 0 \]
Bottom-up Matching

E-matching \textit{wastes a lot of work}

\[
\forall x, \forall y, \forall z, x + (y + z) = (x + y) + z
\]

More efficient to match \textit{bottom-up}

\textbf{Combine} all patterns from quantified equalities

Search for \textbf{leaf nodes first}, recording matching classes

Higher-level patterns \textbf{refer to recorded matches}

RETE algorithm combines patterns into \textbf{state machine}
Complexity

$$(((x + y) + (z + w)) + ((u + v) + (s + t)))$$

**Commutativity** grows the term database... linearly

$$\forall x \forall y, x + y = y + x$$

**Associativity** grows the term database... exponentially

$$\forall x \forall y, (x + y) + z = x + (y + z)$$

**Inversion** grows the term database... infinitely

$$\forall x \forall y, f^{-1}(f(x)) = x$$
Next class: Integers

To do:
- Course feedback
- Read Chapter 9
- Assignment 2
Equality

Turning syntactic equality into **pointer equality**

Pointer sharing in tree structures

**Solving** equality queries with congruence closure

Rebuilding to ensure accuracy

**Model-based** quantifier instantiation

And why the heuristic doesn’t work in general
INEQUALITIES

ELIMINATING VARIABLES
SPEED
LINEAR
OPTIMIZATION
Next class: Integers

To do:
- Course feedback
- Read Chapter 9
- Assignment 2