From Software Verification to Congruence Closure

Christelle Scharff

Collaboration: Dr. Leo Bachmair, SUNY Stony Brook

Implementation: Eugene Kipnis, undergraduate student, Pace University

Support: CSIS summer grant, 2002

Presented: UNIF 2003, Coppenhaguen, Denmark

Faculty Research Day, 05/01/2003
Outline

- Software verification
- Software verification systems
- Equalities
- Word problem
- Abstract congruence closure
- Graph based abstract congruence closure
- Conclusion, Implementation, Future work
Software Verification

- Discover and ascertain properties of programs and a fortiori prove the correctness of a program
- Properties
- Correctness - The program does what we want it to do
- Validation (Testing) versus (Formal) Verification (Proving)
Software Verification

- Verification requires specialized tools:
  - theorem provers
  - deduction systems
  - mathematical expertise
- Software verification systems (PVS, COQ, Isabelle, CAVEAT...)

From Software Verification to Congruence Closure – p.4/22
Why?

- Improvements in software quality
- More than justified in general
- In particular for critical systems (air traffic control, railways signalizing, spacecraft, medical control systems...) where failure must be avoided (economic losses, physical damage or threats in human life).
Equalities

- Software verification systems must reason proficiently about equalities
  - All programs use equalities (assignment and conditions for example)
  - All proofs require reasoning about equalities
- Example: Array indexing
  \[
  (i = j \text{ and } k = l \text{ and } a[i] = b[k] \text{ and } j = a[j] \\
  \text{and } m = b[l]) \implies a[m] = b[k]
  \]
Program

- A program can be seen as a set of equalities.
- Example: Reversing a list

\[
\text{reverse}(\text{nil}) = \text{nil} \\
\text{reverse}(x::L) = \text{reverse}(L)@[x];
\]

- Verifying a program or a property can be seen as reasoning on the structure of the program and as applying structural transformations to the program.
- As transformations are non trivial we end up with a formal method.
Terms

- A term is
  - a variable \((x, y, z\ldots)\)
  - a constant \((a, b, c\ldots)\)
  - \(f(t_1, \ldots, t_n)\) where \(f\) is a \(n\)-ary function symbol and each \(t_i\) is a term.
- A ground term \((f(a), g(c, d)\ldots)\) has no variables.
- \(u[s]\): The term \(u\) contains the term \(s\).
Example

- \( f(a + x, h(f(a, b))) \) is represented by the tree (without sharing):

```
  +
 /|
/a |x
  |
```

```
  f
 /|
/1 |2
+| h
 | |
|/|
/| |
|/|
/a |b
  |
```

```
  f
 /|
/1 |2
+| h
 | |
|/|
/| |
|/|
/a |b
  |
```
Equalities

- \( \approx \) is a congruence relation i.e.
  1. **Reflexive**: \( x \approx x \)
  2. **Symmetric**: \( x \approx y \rightarrow y \approx x \)
  3. **Transitive**: \( x \approx y \) and \( y \approx z \rightarrow x \approx z \)
  4. **Congruence**:
      \( x_1 \approx y_1 \) and \( \ldots \) and \( x_n \approx y_n \)
      \( f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n) \)

- **Examples**: \( \forall x, y, \ (x + y \approx y + x) \) (C),
  \( \forall x, y, z, \ ((x + y) + z) \approx x + (y + z)) \) (A),
  \( \forall x, \ (f(x) \approx a) \)
The Word Problem

- Given a set of equalities $E$ and a goal $s \approx t$, is $s \approx t$ true in all models of $E$?
- The Word problem is undecidable.
- The word problem for ground equalities is decidable [Ackerman,54]

**Examples:**
- $f(f(a,b), b) \approx a$ is a consequence of $f(a,b) \approx a$.
- $f(a) \approx a$ is a consequence of $f(f(f(a))) \approx a$ and $f(f(f(f(f(a))))) \approx a$. 
Focus: Ground Decision Procedures

- Essential to a number of analysis tools for better engineered software including:
  - Array-bound checking
  - Extended static checking
  - Type checking
  - Static analysis
2 Distinct Approaches to Solve the Word Problem in the Ground Case

- Completion
- Congruence Closure
Completion versus Congruence Closure

- Ground Completion [Knuth,Bendix,70] - Special case of Completion that is decidable
  - Compilation of the set of equalities into a set of oriented equalities (called rewriting rules)
- Ground Completion is in general less efficient than Congruence Closure – $nO(\log n)$. 
Completion versus Congruence Closure

- Congruence Closure - of a relation on a graph
  [Kozen, Downey, Sethy, Tarjan, Nelson, Oppen, Bachmair, Tiwari, Shankar, Vigneron]
  - Compact representation of the given terms by a DAG
  - Identifying similarities between patterns in equalities

- Focus: Combination of the 2 approaches
Abstract Congruence Closure

[Bachmair, Tiwari, Vigneron, 00]

- Introduction of constant symbols to abstractly represent sharing (DAG).
- Set of syntactic inference rules to construct the abstract congruence closure (Extension, Simplification, Orientation, Deletion, Deduction, Collapse, Composition)
- A convergent rewrite system over an extended signature $\Sigma \cup \mathcal{K}$. 
“SER” graphs

- To represent the equalities
- Simpler version of SOUR graphs (used for Completion) [Lynch, Strogova, 95].
- Vertex labeled by a symbol of the original signature ($\Sigma$) and a constant ($c_i$) of $\mathcal{K}$.

$$Edges = \begin{cases} 
Subterm & - S \\
Equality & - E \\
Rewriting & - R 
\end{cases}$$

- Each vertex represents a term.
Example

- \( E = \{ f(a, b) \approx a \} \), \( \Sigma = \{ f, a, b \} \)
- \( a \rightarrow c1, b \rightarrow c2, f(c1, c2) \rightarrow c3, c3 \approx c1 \)
SER Abstract Congruence Closure

- ACC is implemented by graph transformations
- Four rules:
  - Orient – An equality,
  - SR, RR – Critical Pairs/Simplification
  - Merge – To ensure closure under congruence.
Orient Rule

\[ c_1 > c_2 \]

\[ V_1 \rightarrow E \rightarrow V_2 \]

\[ V_1 \rightarrow R \rightarrow V_2 \]

\[ c_1 > c_2 \]
Example - Orient - $c_3 > c_1$

$a \rightarrow c_1, \ b \rightarrow c_2, \ f(c_1, c_2) \rightarrow c_3, \ c_3 \rightarrow c_1$
Conclusion and Future Work

- Combination of 2 approaches to solve the Word Problem
- Implementation developed by Eugene Kipnis, undergraduate student at Pace
- Incremental Congruence Closure
- Common framework for the development of reasoning systems
- Software components that capture the basic building blocks of inferences i.e. little engine of proofs