# Sovling The Theory of Equality and Uninterpreted Functions (EUF)

Bruno Andreotti

Universidade Federal de Minas Gerais - UFMG

2024–10–23

# Review: the EUF theory

- Last week, we briefly talked about the theory of equality with uninterpreted functions (EUF)

- Formally, the axioms of this theory are built into the equational first-order logic we use in SMT
  - which is why it is sometimes called the “empty” theory

# Review: the EUF theory

- Last week, we briefly talked about the theory of equality with uninterpreted functions (EUF)

- Formally, the axioms of this theory are built into the equational first-order logic we use in SMT
  - which is why it is sometimes called the “empty” theory

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

# Review: the EUF theory

- Reflexivity: $\forall a.\ \ a = a$
- Symmetry: $\forall a, b.\ \ (a = b) \Leftrightarrow (b = a)$
- Transitivity: $\forall a, b, c.\ \ (a = b) \wedge (b = c) \Rightarrow (a = c)$

# Review: the EUF theory

- Reflexivity: $\forall a.\ \ a = a$

- Symmetry: $\forall a, b.\ \ (a = b) \Leftrightarrow (b = a)$

- Transitivity: $\forall a, b, c.\ \ (a = b) \wedge (b = c) \Rightarrow (a = c)$

- Congruence: $\forall a, b, f.\ \ (a = b) \Rightarrow (f(a) = f(b))$

# Formal definition of the EUF problem

- When solving an SMT problem that involves the EUF theory, the solver will consider all equality terms as atoms, and search for a model

# Formal definition of the EUF problem

- When solving an SMT problem that involves the EUF theory, the solver will consider all equality terms as atoms, and search for a model
- From the perspective of the theory solver, this model will be a series of equalities or disequalities, and we must determine if they are consistent with the EUF axioms or not
- This is equivalent to the problem of finding a “congruence closure” of the equalities, which is the minimal equivalence relation that contains the given equalities, while also respecting the axioms of reflexivity, symmetry, transitivity and congruence.

# Formal definition of the EUF problem

- Therefore, a possible algorithm is to first compute the set of equivalence classes induced by the given set of equalities, and then check if any of the disequalities is between two terms of the same equivalence class

# Formal definition of the EUF problem

- Therefore, a possible algorithm is to first compute the set of equivalence classes induced by the given set of equalities, and then check if any of the disequalities is between two terms of the same equivalence class

- This will be the general shape of the algorithm I will describe today, but first we need to do some pre-processing!

# Currying

- It's complicated to deal with functions that can have any number of arguments
- Instead, we transfrom these terms into their *Curry Form*
- A function $f : (X \times Y) \rightarrow Z$ becomes $f' : X \rightarrow (Y \rightarrow Z)$

# Currying

- It's complicated to deal with functions that can have any number of arguments
- Instead, we transfrom these terms into their *Curry Form*
- A function $f : (X \times Y) \to Z$ becomes $f' : X \to (Y \to Z)$
  ...and an application term $f(x, y)$ becomes $((f'\, x)\, y)$

# Currying

- We are going to introduce a new “apply” function symbol, denoted “$\cdot$”, that takes two arguments
- We represent every function term by applying “$\cdot$” to the function and the argument:

# Currying

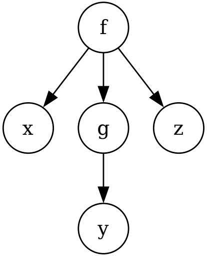
- We are going to introduce a new “apply” function symbol, denoted “$\cdot$”, that takes two arguments

- We represent every function term by applying “$\cdot$” to the function and the argument:

$$\begin{aligned} f(x) &\mapsto \cdot(f, x) \\ f(x_1, \dots, x_n) &\mapsto \cdot(\dots \cdot(\cdot(f, x_1), x_2), \dots x_n) \\ x &\mapsto x \end{aligned}$$

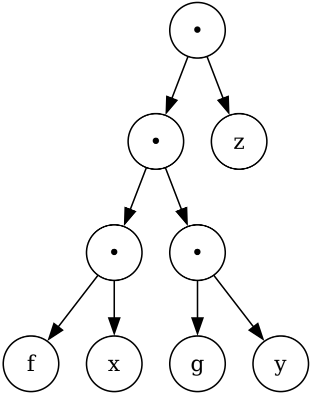
# Currying: Example

$f(x, g(y), z)$



$\Rightarrow$

$\cdot(\cdot(\cdot(f, x), \cdot(g, y)), z)$



# Computing congruence closure

- At the start, each term will be in its own equivalence class
- We will compute the congruence closure by iterating over the input equalities, and merging the equivalence classes as needed
- After processing all equalities, we will have constructed the congruence closure induced by them

# Computing congruence closure

- `pending`: The list of input equalities that we have not yet processed
- `representative(`$t$`)` or $t'$: For each term $t$, this stores a term $r$ which is a unique representative of the equivalence class of $t$. This is also denoted as $t'$. At the start, $t' = t$.
- `class(`$r$`)`: For each representative, stores a list with all the terms in its class.

# Computing congruence closure

- `pending`: The list of input equalities that we have not yet processed
- `representative(`$t$`)` or $t'$: For each term $t$, this stores a term $r$ which is a unique representative of the equivalence class of $t$. This is also denoted as $t'$. At the start, $t' = t$.
- `class(`$r$`)`: For each representative, stores a list with all the terms in its class.
- `lookup(`$a$`, `$b$`)`: For each term $\cdot(a, b)$, `lookup(`$a'$`, `$b'$`)` will a term $c$ such that $c$ is equivalent to $\cdot(a, b)$, if it exists. Otherwise, `lookup` returns `null`.
- `useList(`$r$`)`: For each representative $r$, this stores a list of all terms $\cdot(x, y)$ where $x' = r$ or $y' = r$

# Computing congruence closure

```
function congruence_closure():
    while pending $\neq \varnothing$:
        take $a = b$ from pending
        if $a' \neq b'$:
            merge($a$, $b$)
```

# Computing congruence closure

```
function congruence_closure():
    while pending $\neq \varnothing$:
        take $a = b$ from pending
        if $a' \neq b'$:
            merge($a$, $b$)
```

```
// Assume $class(a') < class(b')$
function merge($a$, $b$):
    for each $c$ in class($a'$):
      set the representative of $c$ to $b'$
      class($b'$) += $c$

    for each $e = \cdot(c, d)$ in useList($a'$):
        let $f$ = lookup($c'$, $d'$)
        if $f \neq$ null and $f' \neq e'$:
            pending += ($e' = f'$)
        lookup($c'$, $d'$) = $e'$
        useList($b'$) += $\cdot(c, d)$)
```

# Complexity analysis

- The `lookup` function can be implemented using a hash table or array, so accessing it is $O(1)$
- Each term can only change representative up to $O(\log n)$ times[1], so the total time spent updating the `representative` table is $O(\log n)$
- Similarly, each input equation $\cdot(c, d) = e$ can only change `useLists` $O(\log n)$ times.
- In total, the complexity to construct the congruence closure is $O(n \log n)$

---

[1] note that the class size always at least doubles after a merge

# Complexity analysis

- We must also consider the complexity of checking if two terms are congruent

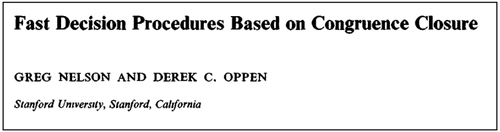
- To do this, we just compute the representatives of each term, and compare them. Since the depth of the tree is bounded $O(\log n)$, this takes $O(\log n)$ time.

- In total, performing the queries to see if the at most $n$ input disequalities are consistent takes $O(n \log n)$ time.

# Complexity analysis

- As such, the total complexity of this algorithm is $O(n \log n)$.
- Notably, this is very cheap compared to most other theory solvers
- The best known algorithms for many common theories are exponential (e.g. quantifier-free linear integer arithmetic), doubly-exponential (e.g. non-linear real arithmetic) or even undecidable (e.g. non-linear integer arithmetic)!

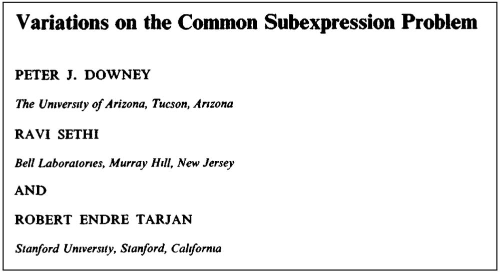
# Some historical context



## Fast Decision Procedures Based on Congruence Closure

**GREG NELSON AND DEREK C. OPPEN**

*Stanford University, Stanford, California*



## Variations on the Common Subexpression Problem

**PETER J. DOWNEY**

*The University of Arizona, Tucson, Arizona*

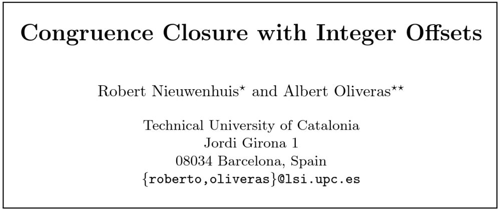
**RAVI SETHI**

*Bell Laboratories, Murray Hill, New Jersey*

**AND**

**ROBERT ENDRE TARJAN**

*Stanford University, Stanford, California*

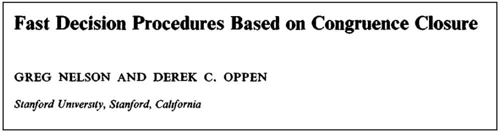


## Congruence Closure with Integer Offsets

Robert Nieuwenhuis[*] and Albert Oliveras[**]

Technical University of Catalonia
Jordi Girona 1
08034 Barcelona, Spain
`{roberto,oliveras}@lsi.upc.es`

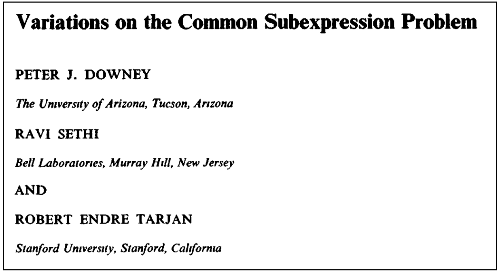
# Some historical context



### Fast Decision Procedures Based on Congruence Closure

**GREG NELSON AND DEREK C. OPPEN**

*Stanford University, Stanford, California*



### Variations on the Common Subexpression Problem

**PETER J. DOWNEY**

*The University of Arizona, Tucson, Arizona*

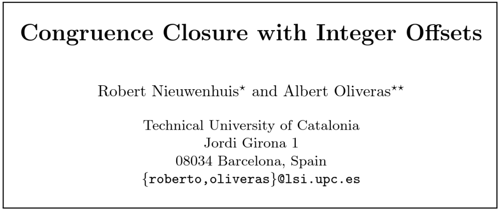
**RAVI SETHI**

*Bell Laboratories, Murray Hill, New Jersey*

**AND**

**ROBERT ENDRE TARJAN**

*Stanford University, Stanford, California*



### Congruence Closure with Integer Offsets

Robert Nieuwenhuis[*] and Albert Oliveras[**]

Technical University of Catalonia
Jordi Girona 1
08034 Barcelona, Spain
`{roberto,oliveras}@lsi.upc.es`

### Proof-producing Congruence Closure

Robert Nieuwenhuis[*] and Albert Oliveras[**]

Technical University of Catalonia, Jordi Girona 1, 08034 Barcelona, Spain
`www.lsi.upc.es/~roberto` `www.lsi.upc.es/~oliveras`

# Proofs and explanations

- So far, we have described an algorithm that can answer **“yes”** or **“no”** to whether two terms are equivalent under congruence

- However, in some cases this boolean answer might not be good enough

# Proofs and explanations

- For example, consider the case where the set of input equalities is:

$$a = f(b), \quad b = c, \quad f(c) \neq a, \quad d_1 = e_1, \quad d_2 = e_2, \quad \ldots, \quad d_n = e_n$$

# Proofs and explanations

- For example, consider the case where the set of input equalities is:

$$a = f(b), \quad b = c, \quad f(c) \neq a, \quad d_1 = e_1, \quad d_2 = e_2, \quad \ldots, \quad d_n = e_n$$

- If we simply tell the SAT solver that this model is inconsistent but don't give any more information, it will create the conflict clause:

$$a \neq f(b) \ \vee \ b \neq c \ \vee \ f(c) = a \ \vee \ d_1 \neq e_1 \ \vee \ d_2 \neq e_2 \ \vee \ \ldots \ \vee \ d_n \neq e_n$$

# Proofs and explanations

- For example, consider the case where the set of input equalities is:

$$a = f(b), \quad b = c, \quad f(c) \neq a, \quad d_1 = e_1, \quad d_2 = e_2, \quad \ldots, \quad d_n = e_n$$

- If we simply tell the SAT solver that this model is inconsistent but don't give any more information, it will create the conflict clause:

$$a \neq f(b) \ \vee \ b \neq c \ \vee \ f(c) = a \ \vee \ d_1 \neq e_1 \ \vee \ d_2 \neq e_2 \ \vee \ \ldots \ \vee \ d_n \neq e_n$$

- In this case, the SAT solver might keep producing very similar inconsistent models

# Proofs and explanations

- Instead, it would be nice if we could convey to the solver that only the first three atoms are enough to form an inconsistent model

- To do this, we will modify our algorithm so that, when we determine that two terms are equivalent, we can also produce an *explanation* of their equivalence

# Proofs and explanations

- Formally, an explanation is a minimal[2] set of equalities that is sufficient to make two terms equivalent
- In the example, the explanation for $f(c) \equiv a$ is $\{a = f(b),\ b = c\}$
- From this, we create the conflict clause

$$a \neq f(b) \ \vee \ b \neq c \ \vee \ f(c) = a$$

which is much more useful

---

[2] as in, if you remove any equality from it, it doesn't work anymore

# Proofs and explanations

- Besides their use as conflict clauses, explanations can also be used to construct *proofs* of unsatisfiability:

$$\dfrac{\dfrac{a = f(b) \qquad \dfrac{b = c}{f(b) = f(c)}\,\text{cong.}}{a = f(c)}\,\text{trans.}}{f(c) = a}\,\text{symm.}$$

# Proofs and explanations

- Besides their use as conflict clauses, explanations can also be used to construct *proofs* of unsatisfiability:

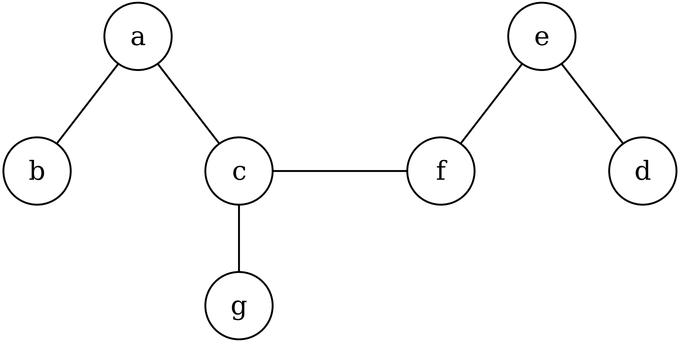
$$\dfrac{\dfrac{a = f(b) \qquad \dfrac{b = c}{f(b) = f(c)}\ \text{cong.}}{a = f(c)}\ \text{trans.}}{f(c) = a}\ \text{symm.}$$

- These are crucial when you want to produce a proof of the unsatisfiability of the formula as a whole

# Proof producing congruence closure

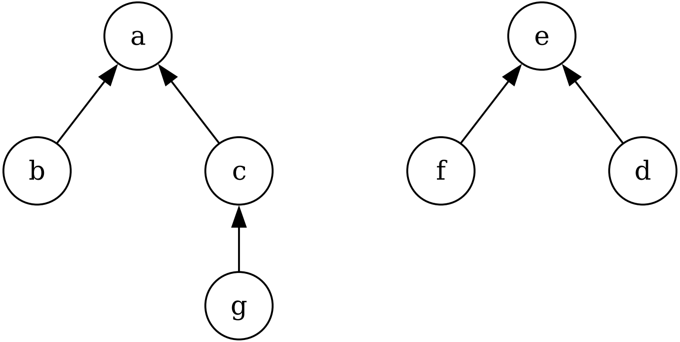
- We will construct a graph where the nodes are terms, and the edges are the class merges that were done

1. $a = b$
2. $c = a$
3. $d = e$
4. $f = e$
5. $g = c$
6. $c = f$



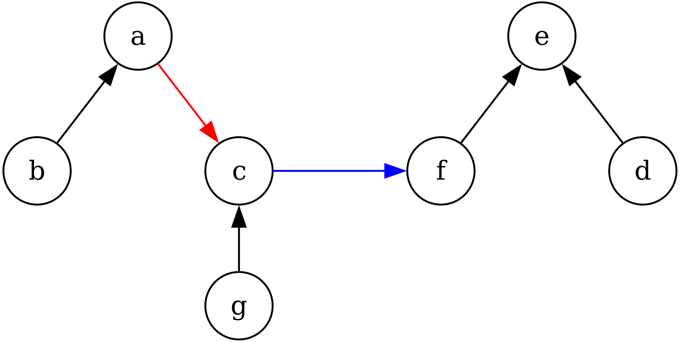
# Proof producing congruence closure

1. $a = b$
2. $c = a$
3. $d = e$
4. $f = e$
5. $g = c$
6. $c = f$



# Proof producing congruence closure

1. $a = b$
2. $c = a$
3. $d = e$
4. $f = e$
5. $g = c$
6. $c = f$



# Proof producing congruence closure

```
function get_explanation($start$, $end$):
    let explanation = []
    let $lca$ = find_lowest_common_ancestor($start$, $end$)
    explanation += explain_along_path($start$, $lca$)
    explanation += explain_along_path($end$, $lca$)
    return explanation
```

# Proof producing congruence closure

```
function explain_along_path($lower$, $upper$):
    let explanation = []
    let $a$ = $lower$
    while $a$ != $upper$:
        let $b$ = parent($a$)
        if the edge $a \rightarrow b$ is $f(a_1, a_2) = f(b_1, b_2)$:
            // the edge is a congruence edge
            explanation += get_explanation($a_1$, $b_1$)
            explanation += get_explanation($a_2$, $b_2$)
        else:
            // the edge is a single input equality
            explanation += ($a = b$)

    return explanation
```

# Proof producing congruence closure

- Here, we've only shown the version of the algorithm that produces explanations

- However, it is not much more complicated to instrument it to also produce structured proofs
  - each input equality you add to the explanation is an assumption to the proof
  - each congruence edge you visit becomes a congruence step
  - and we have to add transitivity steps to connect the path

# Complexity analysis: merge

- With this modified algorithm, we must do some extra work when merging equivalence classes, as we may have to reorient edges up to the root of one of the merged classes.
- Since the proof graph is a forest, we have at most $n-1$ edges, and each edge can be reoriented at most $O(\log n)$ times.
- So, the total time spent doing this extra work is $O(n \log n)$

# Complexity analysis: explain

- The way we implemented `get_explanation` is not the most efficient, as it may try to explain the same terms multiple times

- Solving this limitation is tricky, but can be done with the use of an additional union-find data structure

- With this optimization, the complexity of producing the explanation can be $O(k\,\alpha(k,k))$ (where $k$ is the size of the final proof), which is bound by $O(n \log n)$

# Thanks!

Reminder: there will be no class on Monday, October 28th.
See you all next Wednesday!

[1] Robert Nieuwenhuis and Albert Oliveras. “Congruence Closure with Integer Offsets”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Moshe Y. Vardi and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 78–90. ISBN: 978-3-540-39813-4.

[2] Robert Nieuwenhuis and Albert Oliveras. “Proof-Producing Congruence Closure”. In: *Term Rewriting and Applications*. Ed. by Jürgen Giesl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 453–468. ISBN: 978-3-540-32033-3.

[3] Andreas Fellner, Pascal Fontaine, and Bruno Woltzenlogel Paleo. “NP-completeness of small conflict set generation for congruence closure”. In: *Form. Methods Syst. Des.* 51.3 (Dec. 2017), pp. 533–544. ISSN: 0925-9856. DOI: `10.1007/s10703-017-0283-x`. URL: `https://doi.org/10.1007/s10703-017-0283-x`.

[4] Oliver Flatt et al. “Small Proofs from Congruence Closure”. In: *2022 Formal Methods in Computer-Aided Design (FMCAD)*. 2022, pp. 75–83. DOI: `10.34727/2022/isbn.978-3-85448-053-2_13`.