A Formally Verified Quadratic Unification Algorithm


Computational Logic Group
Dept. of Computer Science and Artificial Intelligence
University of Seville
Introduction

- A case study: using ACL2 to implement and verify a non-trivial algorithm with efficient data structures
  - Implement the algorithm in ACL2, and compare with similar implementations in other languages
  - Explore the main issues encountered during the verification effort
- Unification algorithm on term dags
  - A naive implementation of unification has exponential complexity, both in time and space
  - The implemented algorithm: quadratic time complexity and linear space complexity
- Why this algorithm?
  - Important in many symbolic computation system
  - Reuse previous work
- Note: no formal proofs about the complexity of the algorithm
Unification

- Unification of terms $t_1$ and $t_2$: find (whenever it exits) a most general substitution $\sigma$ such that $\sigma(t_1) = \sigma(t_2)$
- Martelli–Montanari transformation system (acting on unification problems $S; U$)
  
  **Delete:** $\{ t \approx t \} \cup R; U \Rightarrow_u R; U$
  
  **Occur-check:** $\{ x \approx t \} \cup R; U \Rightarrow_u \bot$ if $x \in \mathcal{V}(t)$ and $x \neq t$
  
  **Eliminate:** $\{ x \approx t \} \cup R; U \Rightarrow_u \theta(R); \{ x \approx t \} \cup \theta(U)$
    
    if $x \in X$, $x \notin \mathcal{V}(t)$ and $\theta = \{ x \mapsto t \}$
  
  **Decompose:** $\{ f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \} \cup R; U \Rightarrow_u$
    
    $\{ s_1 \approx t_1, \ldots, s_n \approx t_n \} \cup R; U$
  
  **Clash:** $\{ f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m) \} \cup R; U \Rightarrow_u \bot$
    
    if $n \neq m$ or $f \neq g$
  
  **Orient:** $\{ t \approx x \} \cup R; U \Rightarrow_u \{ x \approx t \} \cup R; U$ if $x \in X$, $t \notin X$

- We defined a particular unification algorithm by choosing:
  - a concrete data structure to represent terms and substitutions
  - a concrete strategy to exhaustively apply the rules of $\Rightarrow_u$
The verification strategy

- Logic of the Process
- Data Structures
- Efficiency Improvements
- Final Theorems
- Execution in ACL2
- Control of the Process
Proving the essential properties of unification

LOGIC OF THE PROCESS → DATA STRUCTURES → EFFICIENCY IMPROVEMENTS

FINAL THEOREMS → EXECUTION IN ACL2 → CONTROL OF THE PROCESS
Martelli–Montanari transformation system

Delete: $\{ t \approx t \} \cup R; U \Rightarrow_u R; U$

Occur-check: $\{ x \approx t \} \cup R; U \Rightarrow_u \perp$ if $x \in \mathcal{V}(t)$ and $x \neq t$

Eliminate: $\{ x \approx t \} \cup R; U \Rightarrow_u \theta(R); \{ x \approx t \} \cup \theta(U)$
if $x \in X$, $x \notin \mathcal{V}(t)$ and $\theta = \{ x \mapsto t \}$

Decompose: $\{ f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \} \cup R; U \Rightarrow_u$
$\{ s_1 \approx t_1, \ldots, s_n \approx t_n \} \cup R; U$

Clash: $\{ f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m) \} \cup R; U \Rightarrow_u \perp$
if $n \neq m$ or $f \neq g$

Orient: $\{ t \approx x \} \cup R; U \Rightarrow_u \{ x \approx t \} \cup R; U$ if $x \in X$, $t \notin X$

- Theorem:
  - If $\{ s = t \}; \emptyset \Rightarrow_u S_1; U_1 \Rightarrow_u \ldots \Rightarrow_u \perp$, the $s$ and $t$ are not
unifiable
  - If $\{ s = t \}; \emptyset \Rightarrow_u S_1; U_1 \Rightarrow_u \ldots \Rightarrow_u \emptyset; U$, then $U$ is a mgu of $s$
and $t$
  - $\Rightarrow_u$ is terminating
Proving the main properties of $\Rightarrow_u$ in ACL2

- Prefix representation of terms and substitutions:
  $$(f \ (h \ z) \ (g \ (h \ x) \ (h \ u)))$$
- We proved the previous theorem, *using the prefix representation of terms*
  - Reasoning is more “natural” with the prefix representation
  - We reused results from other verification projects
- After proving the theorem, in order to verify a concrete unification algorithm, we only have to show that the results computed can be obtained by the application of a sequence of operators of $\Rightarrow_u$
Formalization of $\Rightarrow_u$ in ACL2

- $\Rightarrow_u$ is not a function, is a relation
  - *Operators*: pairs of the form $(name . i)$, where $name$ is one of the rule names
    - $(\text{unif-legal-p upl op})$
    - $(\text{unif-reduce-one-step-p upl op})$
  
- For example:
  
  (defthm mm-preserves-solutions-1
   (implies
    (and (unif-legal-p upl op)
         (solution sigma (both-systems upl)))
    (solution sigma
      (both-systems
       (unif-reduce-one-step-p upl op))))
)
An efficient term representation
Problems with the prefix representation

Exponential behavior

- Problem $U_n$:

  $$p(x_n, \ldots, x_2, x_1) \approx p(f(x_{n-1}, x_{n-1}), \ldots, f(x_1, x_1), f(x_0, x_0))$$

- Mgu: $\{x_1 \leftarrow f(x_0, x_0), x_2 \leftarrow f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$

- With a prefix representation of terms, every application of the Eliminate rule requires reconstruction of the instantiated systems
Unification with term dags

- We represent terms as *directed acyclic graphs (dags)* stored as pointer structures.
- Thus, the **Eliminate** rule only updates a pointer in the graph.
- In ACL2, we represent a graph by the list of its nodes.
- Each node is identified with the index of its position in the list.
Term dags in ACL2

- Example: $f(h(z), g(h(x), h(u))) \approx f(x, g(h(u), v))$
Dag unification problems

- Representing terms as dags, a (sub)term can be identified by the index of its root node
- Dag unification problem: a list \((S \ U \ g)\), where
  - \(g\) is a list of nodes, representing the dag
  - \(S\) and \(U\) system of equations and substitution (resp.) only containing indices, instead of the whole term
- For instance, in the previous example the equation \(g(h(x), h(u)) \approx g(h(u), v)\) is stored as \((4 \ . \ 11)\)
Dag unification

- The key theorem proved in ACL2: the following diagram commutes

\[
\begin{align*}
UPL_p & \xrightarrow{u,p} UPL_p \\
dp & \uparrow \\
UPL_d & \xrightarrow{u,d} UPL_d
\end{align*}
\]

where $\xrightarrow{u,p}$ and $\xrightarrow{u,d}$ denote the transformation relation, defined respectively on prefix unification problems and on dag unification problems.

- The theorem allows us to easily translate the properties proved about $\xrightarrow{u}$, from the prefix representation to the dag representation.
Efficiency improvements

- Logic of the Process
- Data Structures
- Efficiency Improvements
- Final Theorems
- Execution in ACL2
- Control of the Process
Efficiency improvements

- Even with the dag representation the algorithm could be of exponential time complexity. We need to:
  - Improve occur check, avoiding repeated visits to the same subterm
  - Allow sharing of subterms when they have already been unified
- Sharing: after two subterms have been unified, point the root node of one of them to the root node of the other
- We specify this operation *staying at the rule-based level*:
  - Extend $\Rightarrow_{u,d}$ with a new rule: identifications
  - This rule specifies when it is “legal” to do identifications and how it changes the graph
  - But no control issues
A new rule of transformation: identification

- Operator: \((\text{identify } i \ j)\)
- Applicable to a dag unification problem when the subterms pointed by \(i\) and \(j\) are equal
- Results of its application: a new dag unification problem where node \(i\) is updated to point to node \(j\)

**Theorem:** an application of the identification rule does not change the unification problem in prefix form represented by the dag unification problem
Applying the rules with control

LOGIC OF THE PROCESS → DATA STRUCTURES → EFFICIENCY IMPROVEMENTS

FINAL THEOREMS → EXECUTION IN ACL2 ➔ CONTROL OF THE PROCESS

A Formally Verified Quadratic Unification Algorithm – p. 18/32
Applying the rules with control

- Time to define a concrete algorithm: always apply the rule suggested by the first equation
  - And prove that its computation can be simulated by a sequence of applications of $\Rightarrow_{u,d}$ (plus identifications)
- For efficiency reasons, the applicability condition of an identification should not be explicitly checked
  - But the algorithm must arrange things to ensure that whenever an identification is done, the identified subterms are already unified
- We extend the system of equations to be solved with some “identification marks” ($\mathit{id} \ i \ j$)
  - Whenever we apply the \textbf{Decompose} rule to the equation ($i \ . \ j$), we place the identification mark ($\mathit{id} \ i \ j$) just after the equations pairing the arguments of $i$ and $j$
ACL2 implementation: one step of the dag transformation ($\Rightarrow_{u,d}$)

(defun dag-transform-mm-q (ext-dag-upl)
  (let* ((ext-S (first ext-dag-upl)) (equ (first ext-S)) (R (rest ext-S))
         (U (second ext-dag-upl)) (g (third ext-dag-upl)) (stamp (fourth ext-dag-upl))
         (time (fifth ext-dag-upl)))
    (if (equal (first equ) 'id)
        (let ((g (update-nth (second equ) (third equ) g)))
            (list R U g stamp time))
        (let ((t1 (dag-deref (car equ) g)) (p1 (nth t1 g))
               (t2 (dag-deref (cdr equ) g)) (p2 (nth t2 g)))
            (cond ((= t1 t2) (list R U g stamp time))
                ((dag-variable-p p1) (mv-let (oc stamp)
                           (occur-check-q t t1 t2 g stamp time)
                           (if oc nil
                               (let ((g (update-dagi-l t1 t2 g)))
                                   (list R (cons (cons (dag-symbol p1) t2) U) g
                                          stamp (1+ time))))))
                ((dag-variable-p p2) (list (cons (cons t2 t1) R) U g stamp time))
                ((not (eql (dag-symbol p1) (dag-symbol p2))) nil)
                (t (mv-let (pair-args bool)
                           (pair-args (dag-args p1) (dag-args p2))
                           (if bool (list (append pair-args
                                             (cons (list 'id t1 t2) R))
                                      U g stamp time)
                               nil)))))))
ACL2 implementation: one step of the dag transformation ($\Rightarrow_{u,d}$)

dag-transform-mm-q($UPL$) =

let* $UPL$ be $(S$ $U$ $g$ stamp time), $S$ be $(e$ . $R$)
in if first($e$) = id then let $g$ be update-nth(second($e$),third($e$),g)
in ($R$ $U$ $g$ stamp time)  
else let* $t_1$ be dag-deref(car($e$),g), $p_1$ be nth($t_1$,g)
$t_2$ be dag-deref(cdr($e$),g), $p_2$ be nth($t_2$,g)
in if $t_1$ = $t_2$ then ($R$ $U$ $g$ stamp time)
elseif dag-variable-p($p_1$)
let $\langle$oc,stamp$\rangle$ be occur-check-q($t,t_1,t_2,g,$ stamp, time)
in if $oc$ then nil  
else let $g$ be update-nth($t_1,t_2,g$)
in ($R$ $((\text{dag-symbol}(p_1)$ . $t_2)$ . $U$) $g$ stamp time+1)
elseif dag-variable-p($p_2$) then $(((t_2$ . $t_1)$ . $R$) $U$ $g$ stamp time)
elseif dag-symbol($p_1$) $\neq$ dag-symbol($p_1$) then nil
else let $\langle$pair-args,bool$\rangle$ be pair-args(dag-args($p_1$),dag-args($p_2$))
in if $bool$
then ($\text{pair-args}@((\text{id } t_1$ $t_2)$ . $R$) $U$ $g$ stamp time)
else nil
Iteratively applying the rules of $\Rightarrow_u$

(defun solve-upl-q (ext-upl)
  (declare (xargs :measure (unification-measure-q ext-upl)))
  (if (unification-invariant-q ext-upl)
      (if (normal-form-syst ext-upl)
          ext-upl
          (solve-upl-q (dag-transform-mm-q ext-upl)))
      'undef))

- **unification-invariant-q**, a *very long and expensive* condition:
  - Well-formedness
  - Aciclicity
  - Correct placement of the identification marks

- For termination reasons, it has to appear in the body

- Theorem: the computation performed by `solve-upl-q` can be simulated by $\Rightarrow_{u,d}$ (plus identifications)
  - The hard part: show that `unification-invariant-q` is indeed an invariant of the process
Execution in ACL2

- Logic of the Process
- Data Structures
- Efficiency Improvements
- Final Theorems
- Execution in ACL2
- Control of the Process
Execution in ACL2

- The function `solve-upl-q` is executable in ACL2
- But from the practical point of view its execution is completely unfeasible
- For two reasons:
  - Accessing and updating the graph is not done in constant time
  - Expensive well-formedness conditions in the body, needed for termination, and evaluated in every recursive call
Using a stobj to store unification problems

(defstobj terms-dag
  (dag :type (array t (0)) :resizable t)
  ...)

- The stobj allows accessing and updating the graph in constant time
- Single-threadedness is naturally met in this algorithm
- We redefine the algorithm, now with the stobj
- But almost no change from the logical point of view
In general, all the functions traversing the graph are defined using **defexec**.
Execution in ACL2

- Logic of the Process
- Data Structures
- Efficiency Improvements
- Final Theorems
  - Execution in ACL2
  - Control of the Process
Dag unification in ACL2

- The main function `dag-mgu`:
  - Input terms in prefix form are stored as dags in the stobj
  - The Martelli-Montanari transformation rules are exhaustively applied to the dag (updating pointers)
  - If unifiable, the mgu is built from the final dag

- Example:
  - `ACL2 !>(dag-mgu '(f (h z) (g (h x) (h u)))
               '(f x (g (h u) v)))
    (T ((V . (H (H Z))) (U . (H Z)) (X . (H Z))))`
  - `ACL2 !>(dag-mgu '(f y x) '(f (k x) y))
    (NIL NIL)`

- Input and output *in prefix form*, but the main internal operations of the algorithm are performed *with the dag representation*

- The implementation does not use operators (they are only for reasoning)
Main theorems proved

(defthm dag-mgu-completeness
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma)
                       (instance t2 sigma)))
           (first (dag-mgu t1 t2))))

(defthm dag-mgu-soundness
  (let* ((dag-mgu (dag-mgu t1 t2))
         (unifiable (first dag-mgu))
         (sol (second dag-mgu)))
   (implies (and (term-p t1) (term-p t2) unifiable)
            (equal (instance t1 sol) (instance t2 sol)))))

(defthm dag-mgu-most-general-solution
  (let* ((dag-mgu (dag-mgu t1 t2))
         (sol (second dag-mgu)))
   (implies (and (term-p t1) (term-p t2)
                 (equal (instance t1 sigma)
                        (instance t2 sigma))
            (subs-subst sol sigma)))))
### Execution performance

<table>
<thead>
<tr>
<th>$n$</th>
<th>Prefix</th>
<th>Quadratic</th>
<th>C Quadratic</th>
<th>$U_n$</th>
<th>Quadratic</th>
<th>C Quadratic</th>
<th>Prefix</th>
<th>Quadratic</th>
<th>C Quadratic</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0.100</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>4.440</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>13.280</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>20</td>
<td>13.280</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>–</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>–</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>25</td>
<td>–</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>–</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>–</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>30</td>
<td>–</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>–</td>
<td>$\epsilon$</td>
<td>0.001</td>
<td>–</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>100</td>
<td>–</td>
<td>0.002</td>
<td>0.002</td>
<td>–</td>
<td>0.002</td>
<td>0.002</td>
<td>–</td>
<td>0.040</td>
<td>0.032</td>
</tr>
<tr>
<td>500</td>
<td>–</td>
<td>0.052</td>
<td>0.028</td>
<td>–</td>
<td>0.040</td>
<td>0.032</td>
<td>–</td>
<td>0.147</td>
<td>0.138</td>
</tr>
<tr>
<td>1000</td>
<td>–</td>
<td>0.210</td>
<td>0.127</td>
<td>–</td>
<td>0.147</td>
<td>0.138</td>
<td>–</td>
<td>11.591</td>
<td>27.696</td>
</tr>
<tr>
<td>10000</td>
<td>–</td>
<td>75.627</td>
<td>83.047</td>
<td>–</td>
<td>77.856</td>
<td>113.886</td>
<td>–</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Proof effort

<table>
<thead>
<tr>
<th>Phase</th>
<th>Definitions</th>
<th>Theorems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties of $\Rightarrow_u$ (prefix representation)</td>
<td>24</td>
<td>81</td>
</tr>
<tr>
<td>Acyclic graphs</td>
<td>39</td>
<td>101</td>
</tr>
<tr>
<td>Diagram commutativity</td>
<td>39</td>
<td>76</td>
</tr>
<tr>
<td>Storing the initial terms in the graph</td>
<td>29</td>
<td>206</td>
</tr>
<tr>
<td>Extended transformation relation</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>Quadratic improvements and invariant</td>
<td>47</td>
<td>184</td>
</tr>
<tr>
<td>The stobj implementation and guards</td>
<td>26</td>
<td>102</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>214</strong></td>
<td><strong>775</strong></td>
</tr>
</tbody>
</table>
Conclusions

- On the negative side:
  - The number of theorems and definitions needed may be discouraging: 214 definitions and 775 theorems
  - In contrast with a naive implementation (prefix): 19 definitions and 129 theorems
  - Solution: ¿more reusable books?

- On the positive side:
  - The performance of the implementation
  - The successful proof strategy: a rule-based approach clearly separating the logic, the data structures, the control strategy and the ACL2 execution details
  - `mbe` and `defexec` greatly benefits our work