A Theory About First-Order Terms in ACL2

Ruiz-Reina J.L., Alonso, J.A., Hidalgo, M.J., Martín, F.J.

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Introduction

- **We present an ACL2 library formalizing the lattice-theoretic properties of first-order terms**
- **Our purpose is twofold:**
  - theoretical: prove algebraic properties of terms
  - practical: verify some basic algorithms, like matching, renaming, anti–unification and unification
  - these algorithms can be executed in any compliant Common Lisp
- **Example:**
  - Definition and execution:

```
ACL2 !>(anti-unify '(f (h (k u)) x (h y))
                   '(f (h u) (g z) (h z)))
(F (H 3) 2 (H 1))
```

  - Formal properties (greatest lower bound):

```
(defthm anti-unify-lower-bound
  (and (subs (anti-unify t1 t2) t1)
       (subs (anti-unify t1 t2) t2)))

(defthm anti-unify-greatest-lower-bound
   (implies (and (subs term t1)
                 (subs term t2))
            (subs term (anti-unify t1 t2))))
```

- **Usefulness of this library:**
  - Already used in a formalization of term rewriting
  - It could be used to study properties of symbolic computation and automated deduction systems

Representation of first-order terms

- **Terms in prefix notation, using lists:**
  - $f(x, g(y), e)$ **is represented as** `(f x (g y) (e))`
  - **Substitutions as association lists**
- **Useful view: every ACL2 object as a term**
  - **Variables:** `(defun variable-p (x) (atom x))`
  - **Non-variables:** `car` **and** `cdr`**, function symbol and list of arguments, respectively**
- **Recursion for terms and lists of terms**

```
(defun apply-subst (flg sigma term)
  (if flg
      (if (variable-p term)
          (val term sigma)
        (cons (car term)
              (apply-subst nil sigma (cdr term))))
    (if (endp term)
        term
      (cons (apply-subst t sigma (car term))
            (apply-subst nil sigma (cdr term))))))

(defmacro instance (term sigma)
   `(apply-subst t ,sigma ,term))
```

- **A typical example of theorem:**

```
(defthm composition-of-substitutions-apply
  (equal (apply-subst flg (composition sigma1 sigma2) term)
         (apply-subst flg sigma1 (apply-subst flg sigma2 term))))
```

  - **Induction scheme very close to structural induction**
  - **As a particular case, the theorem for terms**
  - **No “type” conditions**

Matching and subsumption

- **Subsumption: $s \leq t$ if and only if $\exists \sigma$ (*matching substitution*) such that $\sigma(s) = t$**

- **The subsumption relation in ACL2**
  - Definition of `(match-mv t1 t2)`, returning two values (a boolean (`subs`) and a substitution (`matching`))
  - The main theorems:

```
(defthm subs-soundness
  (implies (subs t1 t2)
           (equal (instance t1 (matching t1 t2))
                  t2)))

(defthm subs-completeness
  (implies (equal (instance t1 sigma) t2)
           (subs t1 t2)))
```

- **Remark: in order to define a theoretical concept, we defined and verified an executable algorithm `match-mv`, very used in practice**
  - Definition and verification is inspired in a rule-based definition of a unification algorithm (described later)

- **We have proved in ACL2 that the set of terms is a well-founded lattice w.r.t. $\leq$**
  - Well founded quasi-ordering, with *glb* and *lub*
  - We only use the above properties about `subs` and `matching`, defining the subsumption relation

The subsumption quasi-ordering

- **A well-founded quasi-ordering**

```
(defthm subsumption-reflexive (subs t1 t1))

(defthm subsumption-transitive
   (implies (and (subs t1 t2) (subs t2 t3))
            (subs t1 t3)))

(defthm subsumption-well-founded
  (and (e0-ordinalp (subsumption-measure t1))
       (implies (and (subs t1 t2) (not (subs t2 t1)))
                (e0-ord-< (subsumption-measure t1)
                          (subsumption-measure t2)))))
```

- **Equivalent terms and renamings**

```
(defun renamed (t1 t2)
  (and (subs t1 t2) (subs t2 t1)))

(defun renaming (sigma)
  (and (variable-substitution sigma)
       (no-duplicatesp (co-domain sigma))))
```

- **Theorems:**

```
(defthm renaming-implies-renamed
  (implies (and (renaming sigma)
                (subsetp (variables t term)
                         (domain sigma)))
           (renamed (instance term sigma) term)))

(defthm renamed-implies-renaming
  (let ((ren (normal-form-subst t (matching t1 t2) t1)))
    (implies (renamed t1 t2)
             (and (renaming ren)
                  (equal (instance t1 ren) t2)))))
```

A particular renaming

- **For practical purposes, we defined a particular renaming**
  - `(number-rename term x y)`, **which replaces numbers for variables**
  - **Its main property:**

```
(defthm number-renamed-term-renamed-term
  (implies (and (acl2-numberp x) (acl2-numberp y)
                (not (= y 0)))
           (renamed (number-rename term x y) term)))
```

- **Standardization apart**

```
(defthm number-rename-standardization-apart
   (implies (and (acl2-numberp x1) (acl2-numberp x2)
                 (< x1 x2) (< y1 0) (< 0 y2))
            (disjointp
                (variables t (number-rename t1 x1 y1))
                (variables t (number-rename t2 x2 y2)))))
```

- **The** `renamed` **equivalence and congruences**

```
(defequiv renamed)

(defcong renamed iff (subs t1 t2) 1)

(defcong renamed iff (subs t1 t2) 2)
```

- **Congruence rewriting very useful in the mechanization of our proofs**

Greatest lower bound of two terms

- **We define an *anti-unification* algorithm**
  - **Example:**

    ```
    ACL2 !>(anti-unify '(f (h y) x (h y)) '(f (g z) (g z) (g z)))
    (F 1 2 1)
    ```

  - **Auxiliary function** `(anti-unify-aux flg t1 t2 phi)`
  - **By structural recursion, for terms and lists of terms**
  - **The terms are traversed, collecting their common structure**
  - **The argument `phi` is built incrementally, associating numeric variables to corresponding pair of terms with no common structure**

- **Properties of `anti-unify` (lower semilattice):**

```
(defthm anti-unify-lower-bound
  (and (subs (anti-unify t1 t2) t1)
       (subs (anti-unify t1 t2) t2)))

(defthm anti-unify-greatest-lower-bound
   (implies (and (subs term t1)
                 (subs term t2))
            (subs term (anti-unify t1 t2))))
```

- **Proof strategy:**
  - **Incremental construction of `phi`: difficult to prove**
  - **Compositional reasoning: we first verify a similar function, where `phi` is assumed to be *fixed***
  - **Under some conditions on `phi`, this function is equal to `anti-unify`**

Unification of two terms (I)

- **Definitions:**

A substitution σ is a *solution* of a system of equations $S = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ if $\sigma(s_i) \approx \sigma(t_i)$, $1 \leq i \leq n$.

It is a *most general solution* if $\sigma \leq \delta$ for every solution δ of S (where $\sigma \leq \delta$ if there exists a substitution γ such that $\delta = \gamma \circ \sigma$).

A *(most general) unifier* of s and t is a (most general) solution of the system $\{s \approx t\}$.

- **Unification in ACL2**
  - **We defined `(mgu-mv t1 t2)`, returning two values: a boolean (`unifiable`) and a substitution (`mgu`)**
  - **The main theorems:**

```
(defthm mgu-completeness
  (implies (equal (instance t1 sigma)
                  (instance t2 sigma))
           (unifiable t1 t2)))

(defthm mgu-soundness
  (implies (unifiable t1 t2)
           (equal (instance t1 (mgu t1 t2))
                  (instance t2 (mgu t1 t2)))))

(defthm mgu-most-general-unifier
  (implies (equal (instance t1 sigma)
                  (instance t2 sigma))
           (subs-subst (mgu t1 t2) sigma)))
```

  - **Subsumption between substitutions: `subs-sust` (its definition and properties are not trivial)**
  - **The main proof effort of the library**

Unification of two terms (II)

Rule–based specification of unification

Delete:	$\{t \approx t\} \cup R; T$	$\Rightarrow_u R; T$
Decomp:	$\{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow_u \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup R; T$
Conflict:	$\{f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)\} \cup R; T$	$\Rightarrow_u$ `nil` **if** $f \neq g$ **or** $n \neq m$
Orient:	$\{t \approx x\} \cup R; T$	$\Rightarrow_u \{x \approx t\} \cup R; T$ **if** $x \in X$ **and** $t \notin X$
Check:	$\{x \approx t\} \cup R; T$	$\Rightarrow_u$ `nil` **if** $x \in \mathcal{V}(t)$ **and** $x \neq t$
Eliminate:	$\{x \approx t\} \cup R; T$	$\Rightarrow_u \{x \mapsto t\}R; \{x \approx t\} \cup \{x \mapsto t\}T$ **if** $x \in X$ **and** $x \notin \mathcal{V}(t)$

Definition in ACL2

- We define `(transform-mm S T)`, applying one step of transformation with respect to $\Rightarrow_u$
- We define `(solve-system S T bool)`, iteratively applying the transformation rules, until `S` is empty or unsolvability is detected (termination is difficult).
- `mgu-mv` applies `solve-system` to `(list (cons t1 t2))`

Advantages of rule-based specifications:

- Proof clearly separated in two stages (invariants of the transformation steps and termination)
- Logic and control separated (we do not need to specify a concrete selection strategy)
- Nevertheless, some algorithms (anti–unification, for example) are more naturally expressed by recursion on the structure of the terms

Least upper bound of two terms

- Definition of `(mg-instance t1 t2)`
  - Standardize apart `t1` and `t2`
  - Compute a most general unifier (if it exists) of the renamed terms
  - If it exists, apply the unifier to the renamed version of `t1`. Otherwise, return `nil`
  - Examples:

```
ACL2 !>(mg-instance '(f x (h y)) '(f (k u) u))
(F (K (H 1)) (H 1))
ACL2 !>(mg-instance '(f x (h x)) '(f (k u) u))
NIL
```

- Theorems:

```
(defthm common-instance-implies-mg-instance
  (implies (and (subs t1 term) (subs t2 term))
           (mg-instance t1 t2)))

(defthm mg-instance-upper-bound
  (implies (mg-instance t1 t2)
           (and (subs t1 (mg-instance t1 t2))
                (subs t2 (mg-instance t1 t2)))))

(defthm mg-instance-least-upper-bound
  (implies (and (subs t1 term) (subs t2 term))
           (subs (mg-instance t1 t2) term)))
```

Closure properties

- **Terms in a given signature**
  - **Although we have not needed "type conditions", we introduce them to state closure properties**
  - **A general signature**

```
(defstub signat (* *) => *)
```

  - **Well-formed terms in a signature**

```
(defun term-s-p-aux (flg x)
  (if flg
      (if (atom x)
          (eqlablep x)
        (if (signat (car x) (len (cdr x)))
            (term-s-p-aux nil (cdr x))
          nil))
    (if (atom x)
        (equal x nil)
      (and (term-s-p-aux t (car x))
           (term-s-p-aux nil (cdr x))))))

(defmacro term-s-p (x) `(term-s-p-aux t ,x))
```

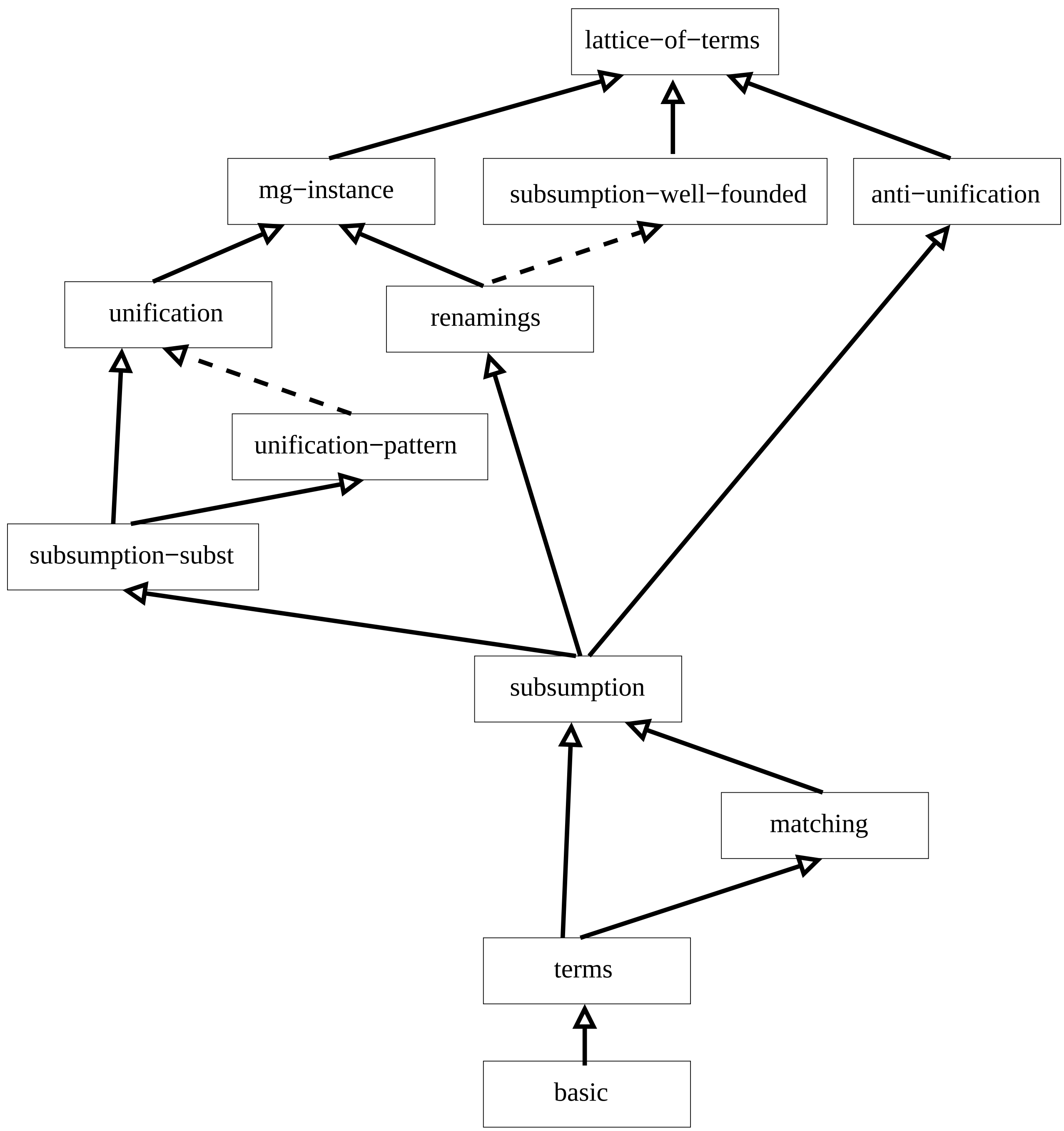
- **The operations defined are closed w.r.t. the terms in a given signature. For example:**

```
(defthm anti-unify-term-s-p
  (implies (and (term-s-p t1) (term-s-p t2))
           (term-s-p (anti-unify t1 t2))))
```

- **As a particular case, the closure properties are used for guard verification**

Conclusions

- **All these properties prove that the set of first-order terms in a given signature (plus an additional top term) is a well-founded lattice with respect to subsumption:**



Conclusions and future work

- **Quantitative information:**

Book	Lines	Definitions	Theorems	Hints
`basic`	378	22	79	2
`terms`	770	53	76	12
`matching`	325	7	48	8
`subsumption`	295	13	29	18
`subsumption-subst`	327	16	38	13
`renamings`	578	9	64	25
`subsumption-well-founded`	216	3	30	7
`anti-unification`	434	10	37	6
`unification-pattern`	808	7	105	33
`unification`	277	12	24	8
`mg-instance`	159	3	17	11
`lattice-of-terms`	148	17	20	5
Total	4715	172	567	148

- **Further work: to improve efficiency of the functions defined, by using better data structures to represent terms**