Verified Computer Algebra in a Computational Logic

Francisco Jesús Martín Mateos

Computational Logic Group

University of Seville

Summary

- Formal verification of programs
- The ACL2 system
- Applying ACL2 to the formal verification of symbolic computation systems
- An example: Dickson's Lemma
- Conclusions

Formal verification of programs

- How to increase our confidence in the *correctness* of a computer program?
- An informal (and classical) approach: *testing and debugging*
  - Problem: It is a sound but not complete method
- A formal approach: *verification*
  - Use mathematical methods to prove that the program meets its intended specification

Formal verification of programs

Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program
(John McCarthy, “A Basis for a Mathematical Theory of Computation” 1961)

- What do we need to formally verify a program?
  - A programming language
  - A logic
  - A theorem prover

Theorem provers

There is a wide spectrum of theorem provers (and proof checkers) that can be used in the formal verification of systems. Their classification can be based in several aspects:

Theorem provers

There is a wide spectrum of theorem provers (and proof checkers) that can be used in the formal verification of systems. Their classification can be based in several aspects:

- Expresiveness of the logic
  - Propositional: zChaff
  - First-Order: ACL2, Otter
  - Set Theory: Mizar
  - Higher-Order: Isabelle/HOL, PVS
  - Type Theory: Coq, Nuprl

Theorem provers

There is a wide spectrum of theorem provers (and proof checkers) that can be used in the formal verification of systems. Their classification can be based in several aspects:

- Expresiveness of the logic
- The programming language
  - With its own language: ACL2
  - Code generated from the specification: Coq, HOL, PVS
  - No programming language: Mizar, Otter

Theorem provers

There is a wide spectrum of theorem provers (and proof checkers) that can be used in the formal verification of systems. Their classification can be based in several aspects:

- Expresiveness of the logic
- The programming language
- Proof automation
  - Automatic: Otter
  - Proof checker: Mizar

Theorem provers

There is a wide spectrum of theorem provers (and proof checkers) that can be used in the formal verification of systems. Their classification can be based in several aspects:

- Expresiveness of the logic
- The programming language
- Proof automation

We will describe in the following the ACL2 system:

- With its own programming language (a subset of Common Lisp)
- Subset of first-order logic (with equality)
- Automatic

The ACL2 system

- ACL2 stands for “**A C**omputational **L**ogic for an **A**pplicative **C**ommon **L**isp”
- Developed in the University of Texas at Austin by J Moore and Matt Kaufmann, since 1994
- Its predecessor is Nqthm, also (well) known as the Boyer-Moore theorem prover
- Successfully used in the industry: verification of hardware components
- But also used in the verification of software and in formalization of mathematics

The ACL2 programming language

- Example:

```
(defun insert (a x)
  (if (consp x)
      (if (<= a (car x))
          (cons a x)
        (cons (car x) (insert a (cdr x))))
    (list a)))

(defun isort (x)
  (if (consp x)
      (insert (car x) (isort (cdr x)))
    nil))
```

The ACL2 programming language

- An applicative subset of Common Lisp
- Applicative:
  - Functions in the language can be seen as functions in the mathematical sense
  - No global variables, no destructive updates
  - No higher-order programming
- Executable in the system (and in any compliant Common Lisp)

```
ACL2 !>(isort '(45 2 34 22/4))
(2 11/2 34 45)
ACL2 !>(isort '(9 8 7 6 5 4 3 2 1 0 -1))
(-1 0 1 2 3 4 5 6 7 8 9)
```

The ACL2 logic

The logic provides the *language* for stating the properties of the defined functions and also a *proof theory* for proving those properties from *axioms* and *definitions*

- Syntax
  - Common Lisp syntax (prefix notation)
  - Propositional connectives and equality: **`and`**, **`or`**, **`not`**, **`implies`**, **`iff`**, **`equal`**
  - Quantifier-free: variables are implicitly universally quantified
  - Example:

```
(defthm isort-correctness
    (and (perm (isort l) l)
         (ordered (isort l))))
```

Axioms and rules of inference

- Axioms:
  - Propositional
  - Equality

    Example: **`(equal x x)`**
  - Primitive Common Lisp functions

    Example: **`(equal (car (cons x y)) x)`**
  - Arithmetic
- Inference rules
  - Propositional
  - Instantiation
  - Proof by induction
- A formula is a *theorem* if it can be derived using the axioms and the rules of inference

Ordinals in ACL2

We can represent in Lisp, by means of dotted pairs and natural numbers, the ordinals below ϵ_0

Ordinal	**ACL2 object**
0	`0`
1	`1`
ω	`((1 . 1) . 0)`
$\omega + 1$	`((1 . 1) . 1)`
$\omega + 2$	`((1 . 1) . 2)`
$\omega 2$	`((1 . 2) . 0)`
$\omega 3$	`((1 . 3) . 0)`
ω^2	`((2 . 1) . 0)`
ω^3	`((3 . 1) . 0)`
ω^ω	`((((1 . 1) . 0) . 1) . 0)`
$\omega^\omega + \omega^2 4 + 3$	`((((1 . 1) . 0) . 1) (2 . 4) . 3)`
$\omega^{(\omega^\omega)}$	`((((((1 . 1) . 0) . 1) . 0) . 1) . 0)`
$\ldots$	...

Well-foundedness in ACL2

- A relation $<$ on a set A is well-founded if there is no infinitely descending chain $a_1 > a_2 > a_3 \ldots$
- The predefined functions **o-p** and **o<**, respectively define the ACL2 ordinals and the usual order between ordinals
- (Meta) Assumption: **o<** is well-founded on **o-p**
- Ordinals are essential to prove properties by induction
- And also in the definition of new functions

Defining new functions

- The logic is not static: a new (definitional) axiom is introduced whenever a new function is defined
  - Example: the definition

```
(defun isort (x)
  (if (consp x)
      (insert (car x) (isort (cdr x)))
    nil))
```

is introduced in the logic as the axiom

```
(equal (isort x)
       (if (consp x)
           (insert (car x) (isort (cdr x)))
         nil))
```

Defining new functions

- The logic is not static: a new (definitional) axiom is introduced whenever a new function is defined
- In order to avoid inconsistencies, we have to prove termination of recursive definitions, by *showing* an ordinal measure on the arguments and *proving* that this measure decrease in every recursive call
  - Example:

```
(defun isort (x)
  (if (consp x)
      (insert (car x) (isort (cdr x)))
    nil))
```

    the (ordinal) measure **`(len x)`** justifies termination of **`isort`**:

```
(and (o-p (len x))
     (implies (consp x)
              (o< (len (cdr x)) (len x))))
```

Defining new functions

- The logic is not static: a new (definitional) axiom is introduced whenever a new function is defined
- In order to avoid inconsistencies, we have to prove termination of recursive definitions, by *showing* an ordinal measure on the arguments and *proving* that this measure decrease in every recursive call
- So in the ACL2 logic all functions are *total*

Induction principle in ACL2

- A particular case of well-founded induction
- Roughly speking, to prove a theorem, we can assume the theorem true for a *finite* number of instances, whenever those instances are proved to be smaller w.r.t. a given ordinal measure
- For example, to prove the property **(ordered (isort l))**, it suffices to prove the formulas

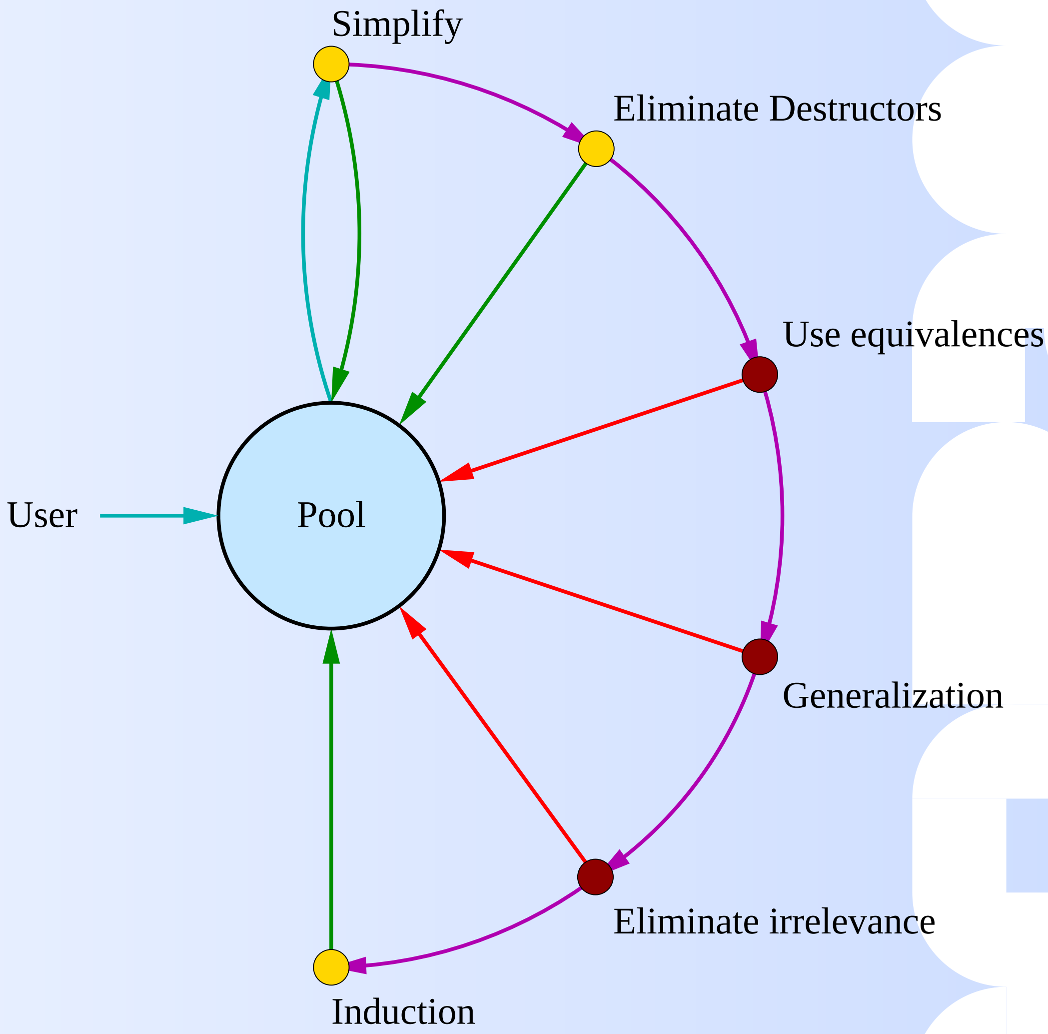
```
(implies (not (consp l)) (ordered (isort l)))

(implies (and (consp l)
              (ordered (isort (cdr l))))
         (ordered (isort l)))
```

- How to find a suitable induction scheme for proving a property?
- Recursive definitions “suggest” induction schemes

The ACL2 theorem prover

- Supports mechanized reasoning in the logic
- Instead of constructing the proof by elementary steps, it tries larger steps
- Six transformation proccesses are tried in order for every (sub)goal formula
- Automatic: Once a conjecture is submitted, the user can no longer interact with the system



Theorem prover output (example)

```
ACL2 !>(defthm isort-correctness
         (and (perm (isort l) l)
              (ordered (isort l))))

We will induct according to a scheme suggested by (ISORT L).  This
suggestion was produced using the :induction rule ISORT.
...
When applied to the goal at hand the above induction scheme produces
the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (NOT (CONSP L))
         (AND (PERM (ISORT L) L)
              (ORDERED (ISORT L)))).
....
```

Theorem prover output (example)

```
...

Subgoal *1.1/1
(IMPLIES (AND (CONSP IT) (<= L1 (CAR IT))
              (PERM IT L2) (ORDERED IT))
         (PERM (INSERT L1 IT) (CONS L1 L2))).

But simplification reduces this to T, using the :definitions DELETE-
ONE, INSERT, MEMBER, ORDERED and PERM, primitive type reasoning, the
:rewrite rules CAR-CONS and CDR-CONS and the :type-prescription rules
MEMBER and PERM.

That completes the proofs of *1.1 and *1.

Q.E.D.
...
```

The role of the user

- Very often, non trivial results fails to be proved in a first attempt
- This means that the prover needs to prove previous lemmas that have to be supplied by the user
- This lemmas are suggested from:
  - A preconceived hand proof
  - Inspection of failed proofs
- Thus, the role of the user is:
  - To formalize the conjectures in the logic
  - Implement a *proof strategy*, by means of a suitable collection of lemmas
- The result of a proof effort is a file with definitions and theorems
  - *A book* in the ACL2 terminology
  - This book can be *certified* and used by other books

The main ACL2 application: hardware verification

- Example: formal verification of the microcode of the division algorithm on the AMD-K5 microprocessor

```
(defun divide (p d mode)
  .......
  ;;; hundreds of lines faithfully reflecting the microcode
  .......)
```

- Correctness theorem:

```
(defthm AMD-K5-division-correct
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
           (equal (divide p d mode)
                  (round (/ p d) mode))))
```

Can we formally verify symbolic computation systems?

- Formal proofs ensuring the correctness of the implemented algorithms are difficult, because:
  - Software systems are much more complicated than hardware systems
  - And in this case the underlying mathematical theory is much richer
- In the Computational Logic Group of the University of Seville, we tried a first step:
  - Verification of basic algorithms of theorem proving and symbolic computation systems
- For example:
  - Equational: rewriting theory, unification, Knuth-Bendix
  - Propositional: tableaux, resolution, Davis-Putnam
  - Polynomials: Gröbner bases computation

Why do we use ACL2 for this task?

- We have computation and deduction in the same system
- And the programming language is Common Lisp
- A major example: *“Formal verification of Buchberger’s algorithm”*, PhD Thesis, Inmaculada Medina Bulo
- The price to pay: the expressiveness of the logic is limited
  - Sometimes is difficult to state mathematical properties
- A detailed example: *“Formal Proof of Dickson’s Lemma in ACL2”*

Buchberger algorithm, a basic implementation

```
(defun Buchberger-aux (F C)
  (declare (xargs :well-founded-relation ??? :measure ???))
   ....
      (if (endp C)
          F
        (let* ((p (first (first C)))
               (q (second (first C)))
               (h (red-F* (s-polynomial p q) F)))
          (if (equal h (zero-polynomial))
              (Buchberger-aux F (rest C))
            (Buchberger-aux (cons h F)
                            (append (pairs h F) (rest C))))))
    ...)

(defun Buchberger (F)
  (Buchberger-aux F (initial-pairs F)))
```

Dickson's lemma

Let $n \in \mathbb{N}$ and $\{m_k : k \in \mathbb{N}\}$ be an infinite sequence of monomials in the variables $\{X_1, \ldots, X_n\}$. Then, there exist indices $i < j$ such that m_i divides m_j.

- Dickson's lemma ensure termination of Buchberger's algorithm
- Difficult to formalize in the ACL2 logic
  - Classical proofs are non-constructive
  - Absence of existential quantifiers: i and j has to be explicitly given
- Representation:
  - Monomials as n-tuples (represented as lists in ACL2): **`tuple-p`**
  - Divisibility as component-wise usual order on naturals: **`tuple-<=`**

Formalization of Dickson's lemma in ACL2

- An arbitrary sequence **f** of **N**-tuples:

```
(encapsulate
 (((N) => *)
  ((f *) => *))

 ...

 (defthm N-is-nat->-0
   (and (integerp (N)) (< 0 (N))))

 (defthm f-provides-N-tuples
   (implies (natp i)
            (and (equal (len (f i)) (N))
                 (tuple-p (f i)))))

 )
```

Formalization of Dickson's lemma in ACL2

- A function computing the indices i and j of Dickson's lemma:

```
(defun get-tuple-<=-f (j T0)
  (if (natp j)
      (cond ((= j 0) nil)
            ((tuple-<= (f (- j 1)) T0) (- j 1))
            (t (get-tuple-<=-f (- j 1) T0)))
    nil))

(defun dickson-indices (j)
  (if (natp j)
      (let ((i (get-tuple-<=-f j (f j))))
        (if i
            (list i j)
          (dickson-indices (+ j 1))))
    nil))
```

Formalization of Dickson's lemma in ACL2

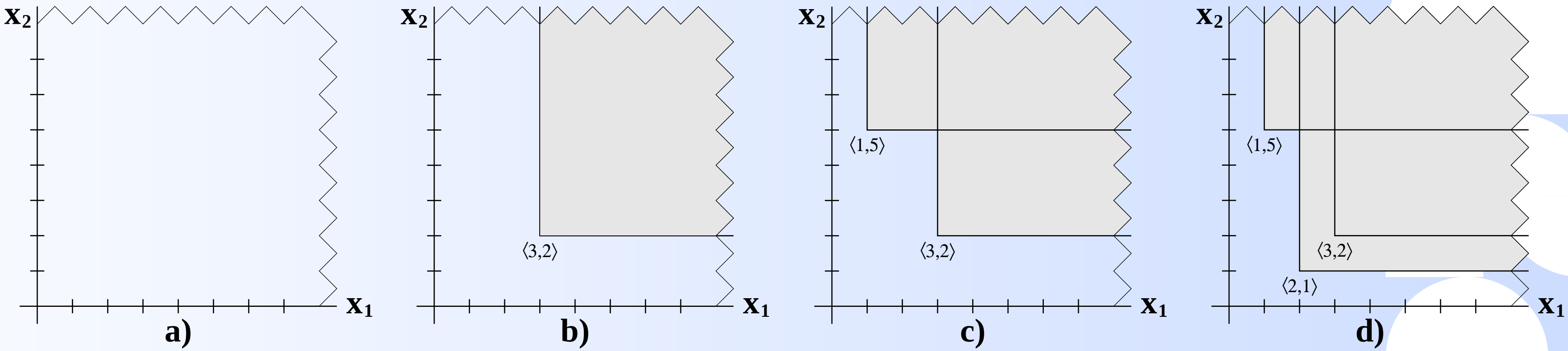
- Assuming termination of **`dickson-indices`**, it is easy to prove:

```
(defthm dickson-lemma
  (implies (natp k)
           (let ((i (first (dickson-indices k)))
                 (j (second (dickson-indices k))))
             (and (< i j) (tuple-<= (f i) (f j))))))
```

- So the main task is to prove termination of **`dickson-indices`**
- Termination of functions in ACL2
  - A well-founded relation
  - A measure of the arguments, taking values in the domain of the relation
  - Prove that the measure decreases w.r.t. the well-founded relation in every recursive call

An intuitive idea of the measure

- Example: assume that $\boldsymbol{f_0 = \langle 3, 2\rangle}$, $\boldsymbol{f_1 = \langle 1, 5\rangle}$ and $\boldsymbol{f_2 = \langle 2, 1\rangle}$



- The “free space” (non-shaded region) decreases in a well-founded way
  - it represents the set of “allowable” tuples at position $\boldsymbol{k}$
  - for every $\boldsymbol{k}$, our measure will “represent” the “size” of this region

Patterns and multiset of patterns

- A pattern is a tuple in $(\mathbb{N} \cup \{*\})^n$
  - Examples: $\langle *, *, * \rangle$, $\langle 5, *, 1 \rangle$, $\langle *, *, 8 \rangle$, $\langle 15, 9, 10 \rangle$
  - It represents all tuples obtained replacing occurrences of $*$ by natural numbers (“hyperplanes” in $\mathbb{N}^n$)
  - Dimension of a pattern: number of $*$’s
- The “free space” can be represented by multiset of patterns and measured by the multiset of the dimensions of the patterns
  - Example: $\{\!\{\langle 0, * \rangle, \langle 1, * \rangle, \langle 2, * \rangle, \langle *, 0 \rangle, \langle *, 1 \rangle\}\!\}$, measured as $\{\!\{1, 1, 1, 1, 1\}\!\}$
- Main idea:
  - Every time a new tuple in the sequence is not divisible by any of the previous tuples, the measure of the remaining “free space” decreases w.r.t. the multiset relation induced by the usual order between natural numbers

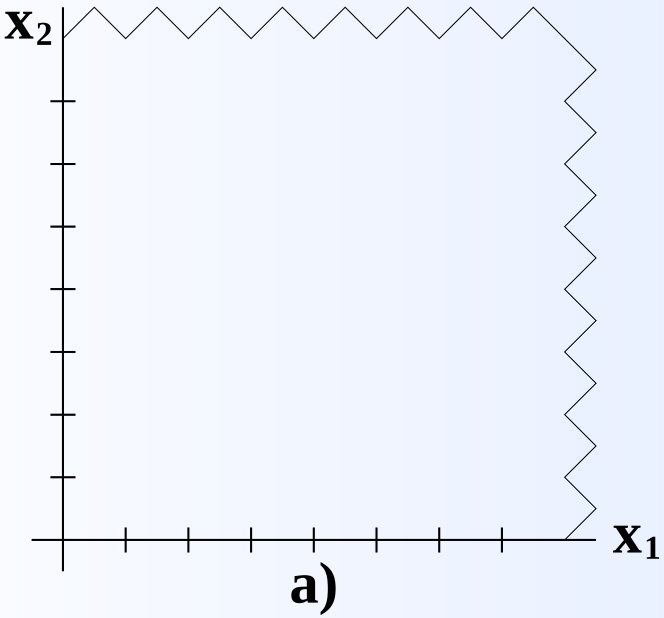
Multiset well-founded relations in ACL2

- Well-founded relations in ACL2
  - Only one predefined well founded relation: **`o<`** on **`o-p`**
  - The user may define well-founded relations: giving a monotone ordinal inmersion
- Induced multiset relations:

$$\{8, 6, 6, 6, 6, 3, 3, 3, 3, 3, 1\} <_{\mathcal{M}} \{8, 8, 6, 5, 3, 1\}$$

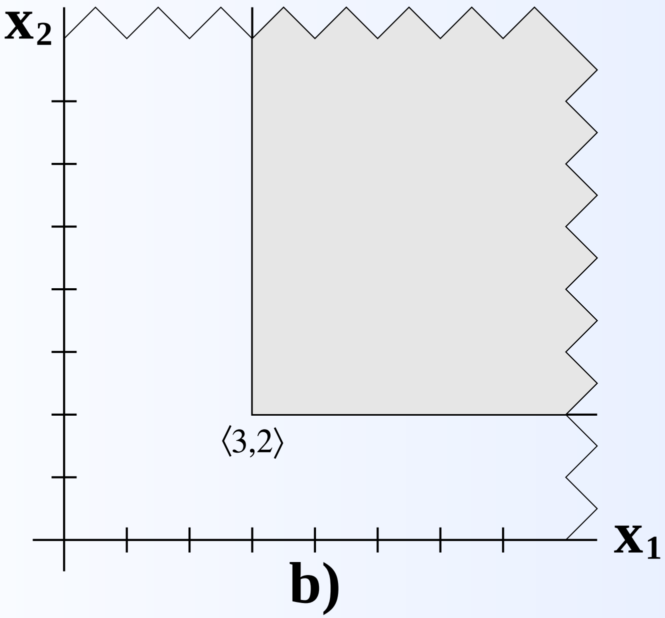
- Automating the generation of well-founded multiset relations:
  `(defmul (O< NIL O-P O<-FN NIL NIL))`
- This call *automatically* defines **`mul-o<`** as the multiset relation induced by **`o<`** on lists of ACL2 ordinals and proves its well-foundedness

Graphical example



- Patterns:
  $\{\!\{\langle *, *\rangle\}\!\}$
- Measure:
  $\{\!\{2\}\!\}$
- Sequence:

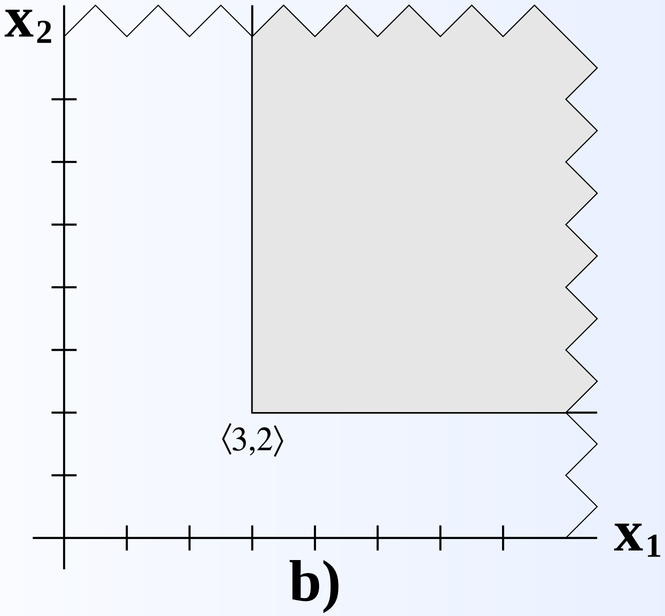
Graphical example



- Patterns:
  $\{\!\{\langle *, * \rangle\}\!\}$

- Measure:
  $\{\!\{2\}\!\}$
- Sequence:
  $f_0 = \langle 3, 2 \rangle$

Graphical example



- Patterns:
  $\{\!\{\langle 0, * \rangle, \langle 1, * \rangle, \langle 2, * \rangle, \langle *, 0 \rangle, \langle *, 1 \rangle\}\!\}$
- Measure:
  $\{\!\{1, 1, 1, 1, 1\}\!\}$
- Sequence:
  $f_0 = \langle 3, 2 \rangle$

Reduction of patterns

- Reducing a pattern by a given tuple:

```
(defun reduce-pattern (T0 P)
  (cond ((endp P) nil)
        ((natp (car P))
         (cons-list-cdr (car P)
                        (reduce-pattern (cdr T0) (cdr P))))
        (t (append (cons-list-car (nat-<-list (car T0)) (cdr P))
                   (cons-list-cdr (car P)
                     (reduce-pattern (cdr T0) (cdr P)))))))
```

Properties of the reduction process

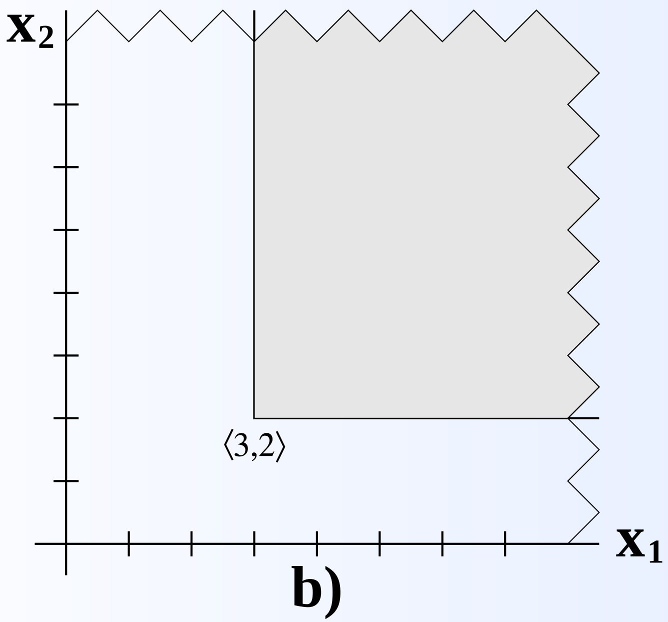
- The measure decreases in the reduction process

```
(defthm reduce-pattern-property
  (implies (member-equal P1 (reduce-pattern T0 P2))
           (o< (pattern-measure P1) (pattern-measure P2))))
```

- The reduction process preserves valid tuples

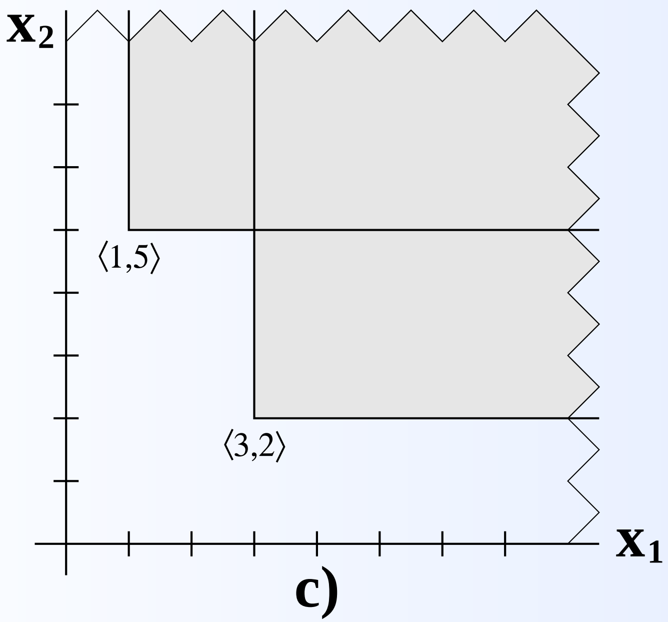
```
(defthm member-pattern-list-reduce-pattern
  (implies (and (member-pattern T1 P)
                (member-pattern T2 P)
                (tuple-p T1)
                (tuple-p T2)
                (not (tuple-<= T1 T2)))
           (member-pattern-list T2 (reduce-pattern T1 P))))
```

Graphical example



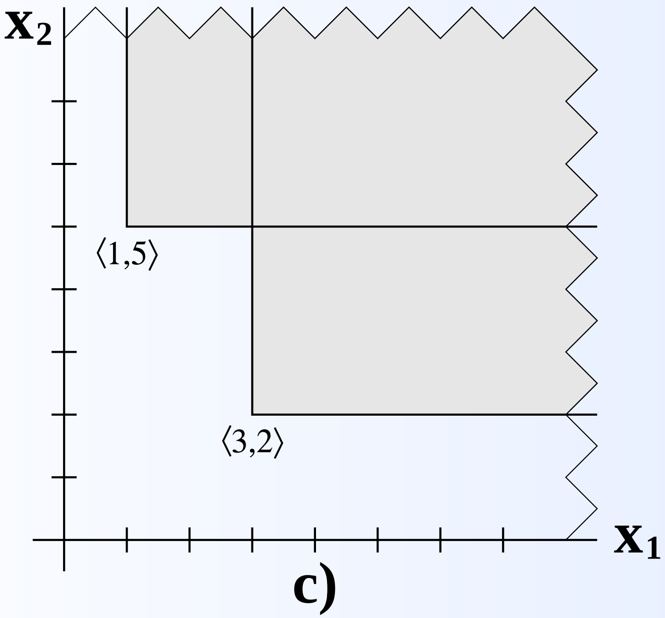
- Patterns:
  $\{\!\{\langle 0, *\rangle, \langle 1, *\rangle, \langle 2, *\rangle, \langle *, 0\rangle, \langle *, 1\rangle\}\!\}$
- Measure:
  $\{\!\{1, 1, 1, 1, 1\}\!\}$
- Sequence:
  $f_0 = \langle 3, 2\rangle$

Graphical example



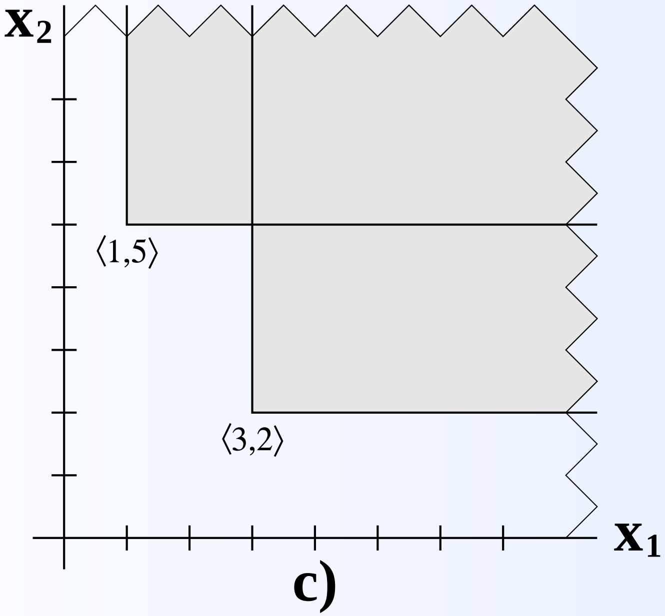
- Patterns:
  $\{\!\{\langle 0,*\rangle, \langle 1,*\rangle, \langle 2,*\rangle, \langle *,0\rangle, \langle *,1\rangle\}\!\}$
- Measure:
  $\{\!\{1,1,1,1,1\}\!\}$
- Sequence:
  $f_0 = \langle 3,2\rangle, f_1 = \langle 1,5\rangle$

Graphical example



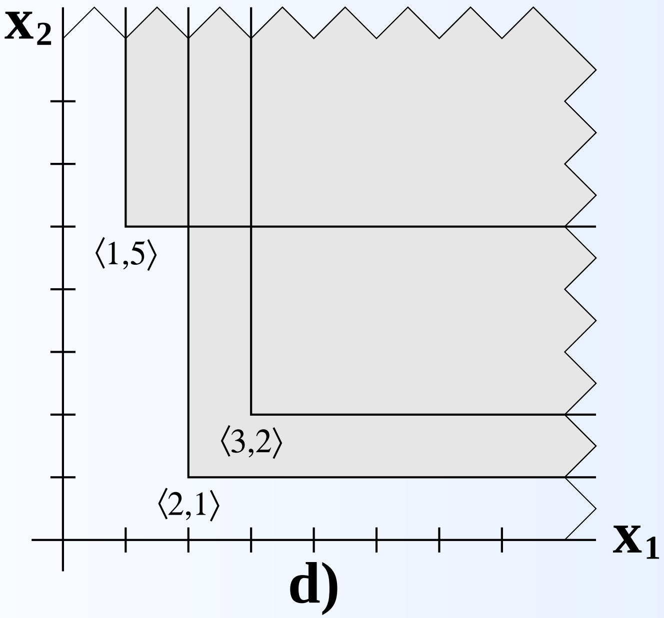
- Patterns:
  $\{\!\{\langle 0, *\rangle, \langle 1, *\rangle, \langle 2, *\rangle, \langle *, 0\rangle, \langle *, 1\rangle\}\!\}$
- Measure:
  $\{\!\{1, 1, 1, 1, 1\}\!\}$
- Sequence:
  $f_0 = \langle 3, 2\rangle, f_1 = \langle 1, 5\rangle$

Graphical example



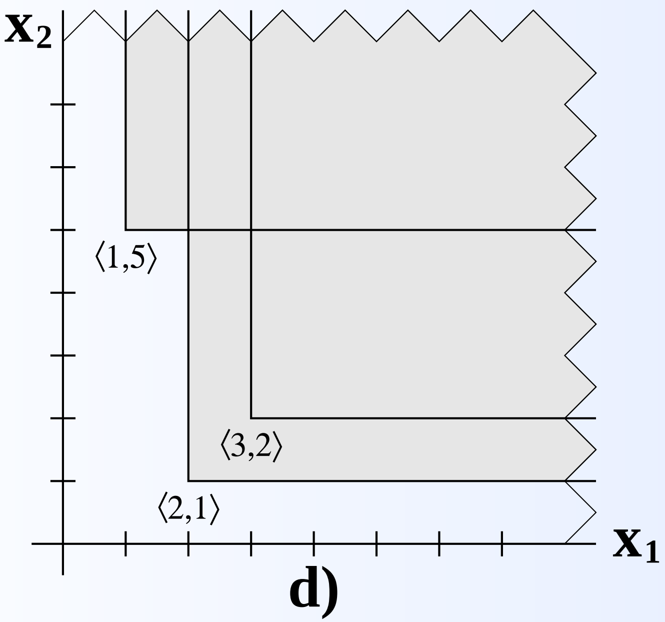
- Patterns:
  $\{\!\{\langle 0,*\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,*\rangle, \langle *,0\rangle, \langle *,1\rangle\}\!\}$
- Measure:
  $\{\!\{1,0,0,0,0,0,1,1,1\}\!\}$
- Sequence:
  $f_0 = \langle 3,2\rangle, f_1 = \langle 1,5\rangle$

Graphical example



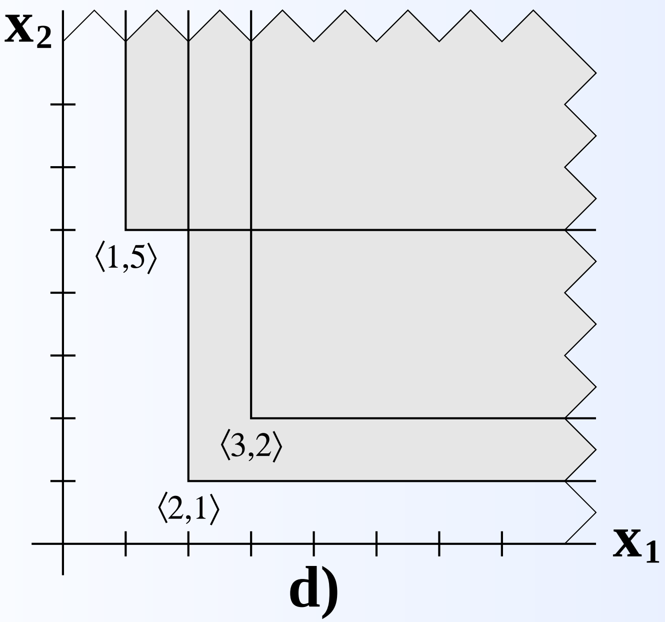
- Patterns:
  $\{\!\{\langle 0, *\rangle, \langle 1, 0\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, *\rangle, \langle *, 0\rangle, \langle *, 1\rangle\}\!\}$
- Measure:
  $\{\!\{1, 0, 0, 0, 0, 0, 1, 1, 1\}\!\}$
- Sequence:
  $f_0 = \langle 3, 2\rangle, f_1 = \langle 1, 5\rangle, f_2 = \langle 2, 1\rangle$

Graphical example



- Patterns:
  $\{\!\{\langle 0, *\rangle, \langle 1, 0\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, *\rangle, \langle *, 0\rangle, \langle *, 1\rangle\}\!\}$
- Measure:
  $\{\!\{1, 0, 0, 0, 0, 0, 1, 1, 1\}\!\}$
- Sequence:
  $f_0 = \langle 3, 2\rangle, f_1 = \langle 1, 5\rangle, f_2 = \langle 2, 1\rangle$

Graphical example



- Patterns:
  $\{\!\{\langle 0,*\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,0\rangle, \langle *,0\rangle, \langle *,1\rangle\}\!\}$
- Measure:
  $\{\!\{1,0,0,0,0,0,0,1,1\}\!\}$
- Sequence:
  $f_0 = \langle 3,2\rangle, f_1 = \langle 1,5\rangle, f_2 = \langle 2,1\rangle$

Reduction of patterns

- Computing the reduced “free space”

```
(defun reduce-pattern-list (T0 Ps)
  (cond ((endp Ps) Ps)
        ((member-pattern T0 (car Ps))
         (append (reduce-pattern T0 (car Ps))
                 (cdr Ps)))
        (t (cons (car Ps)
                 (reduce-pattern-list T0 (cdr Ps))))))
```

Properties of the reduction process

- The measure decreases in the reduction process

```
(defthm reduce-pattern-list-reduces-pattern-list-measure
  (implies (member-pattern-list T0 Ps)
           (mul-o< (pattern-list-measure
                                      (reduce-pattern-list T0 Ps))
                   (pattern-list-measure Ps))))
```

- The reduction process preserves valid tuples

```
(defthm member-pattern-list-reduce-pattern-list
  (implies (and (member-pattern-list T2 Ps)
                (not (tuple-<= T1 T2))
                (tuple-p T1)
                (tuple-p T2))
           (member-pattern-list T2 (reduce-pattern-list T1 Ps))))
```

Reduction of patterns

- Iterating the reduction proccess over a finite sequence of tuples

```
(defun reduce-pattern-tuple-list (Ts Ps)
  (cond ((endp Ts) Ps)
        (t (reduce-pattern-list
             (car Ts)
             (reduce-pattern-tuple-list (cdr Ts) Ps)))))
```

Definition of the measure in ACL2

- The measure:

```
(defun dickson-indices-measure (k)
  (pattern-list-measure (reduce-pattern-tuple-list
                         (initial-segment-f (- k 1))
                         (list (initial-pattern (N))))))
```

  - where **(initial-pattern (N))** $= \langle *, \ldots, * \rangle$,
  - and **(initial-segment-f k)**$=$**(** f_k **...** f_1 f_0 **)**

Final steps

- The main theorem:

```
(defthm dickson-indices-termination-property
  (implies (and (natp k)
                (not (get-tuple-<=-f k (f k))))
           (mul-o< (dickson-indices-measure (+ 1 k))
                   (dickson-indices-measure k))))
```

- Having proved this theorem, **`dickson-indices`** is proved to terminate, taking:
  - **`mul-o<`** as well-founded relation
  - **`dickson-indices-measure`** as measure
- And **`dickson-lemma`** is now easily proved
- Using similar techniques, we have also obtained a mechanized proof of Higman's lemma in ACL2

Conclusions: on the positive side

- Correctness is an important issue for software development
  - Specially if the software is used in critical applications
- Benefits in using ACL2
  - Computing and deduction in the same system
  - Common Lisp
- A formal proof of a mathematical result is interesting in its own
  - Detail, rigor and clarity
  - Deeper understanding

Conclusions: on the negative side

- Nowadays, a fully verified state-of-the-art symbolic computation system seems unfeasible
  - Only the verification of Buchberger algorithm needed almost 300 definitions and 1000 theorems
  - The underlying mathematical theory is complex
- Efficiency has not been our main concern
  - The more sophisticated is the implementation and the data structures used, the more complex is the formal verification

Future work

- Verification of programs with more efficient data structures
- An alternative to program verification: output verification
- Sharing mathematical knowledge
  - A verification effort benefits from other verification efforts
  - Hopefully, with an increasing library of formalized mathematics, easily accesible and portable, this verification effort would be reduced