

# *Mechanized Proof of Buchberger's Algorithm in ACL2*

J.L. Ruiz-Reina, I. Medina and F. Palomo

Departamento de Ciencias de la Computación e Inteligencia Artificial

Univ. Sevilla

Departamento de Lenguajes y Sistemas Informáticos

Univ. Cádiz

## Summary

---

- The ACL2 system
- Applying ACL2 to the formal verification of symbolic computation systems
- An example: Buchberger's algorithm
- Conclusions

## The ACL2 system

---

- ACL2 stands for “**A Computational Logic for an Applicative Common Lisp**”
- 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: hardware verification
- But also used in the verification of software and in formalization of mathematics

## 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)

## 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?

## 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 (for writing the program)

## 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 (for writing the program)
  - A logic (for stating and proving its 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 (for writing the program)
  - A logic (for stating and proving its specification)
  - A theorem prover (to assist in the proof process)

## 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 (for writing the program)
  - A logic (for stating and proving its specification)
  - A theorem prover (to assist in the proof process)

We have these three components in ACL2

# 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))))
```
  - We use *the same language* for both programming and specifying

## 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  $\epsilon_0$

<u>Ordinal</u>	<u>ACL2 object</u>
0	0
1	1
2	2
3	3
...	...
$\omega$	((1 . 1) . 0)
$\omega + 1$	((1 . 1) . 1)
$\omega + 2$	((1 . 1) . 1)

## Ordinals in ACL2

We can represent in Lisp, by means of dotted pairs and natural numbers, the ordinals below  $\epsilon_0$

Ordinal

ACL2 object

...

...

$\omega^2$

((1 . 2) . 0)

...

...

$\omega^2$

((2 . 1) . 0)

...

...

$\omega^2 + \omega^4 + 3$

((2 . 1) (1 . 4) . 3)

...

...

$\omega^3$

((3 . 1) . 0)

## Ordinals in ACL2

We can represent in Lisp, by means of dotted pairs and natural numbers, the ordinals below  $\epsilon_0$

Ordinal

ACL2 object

...

$\omega^\omega$

...

(( ( (1 . 1) . 0) . 1) . 0)

...

$\omega^\omega + \omega^{99}4 + 3$

...

(( ( (1 . 1) . 0) . 1) (99 . 1) . 3)

...

$\omega^{(\omega^\omega)}$

...

(( ( ( ( (1 . 1) . 0) . 1) . 0) . 1) . 0)

...

...

## A built-in notion in ACL2: Well-foundedness

---

- A relation  $<$  on a set  $A$  is well-founded if there is no infinitely descending chain  $a_1 > a_2 > a_3 \dots$
- The predefined functions  $\circ\text{-p}$  and  $\circ<$ , respectively define the ACL2 ordinals and the usual order between ordinals
- (Meta) Assumption:  $\circ<$  is well-founded on  $\circ\text{-p}$
- 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 (either as part of the program or as part of the specification)
  - 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

---

- 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`, since it can be proved that:

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

- So in the ACL2 logic all functions are *total*

## Induction principle in ACL2

---

- A particular case of well-founded induction
- Roughly speaking, 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 1))`, it suffices to prove the formulas

```
(implies (not (consp 1)) (ordered (isort 1)))
```

```
(implies (and (consp 1)  
              (ordered (isort (cdr 1))))  
         (ordered (isort 1)))
```

- The problem is to find a suitable induction scheme for proving a property
- Recursive definitions “suggest” induction schemes

## The ACL2 theorem prover (output example)

---

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

By case analysis we reduce the conjecture to the following two conjectures.

...

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)))).
```

....

## The ACL2 theorem prover (output example)

---

....

Subgoal \*1.1/2

```
(IMPLIES (AND (CONSP IT) (< (CAR IT) L1)
              (NOT (PERM (CDR IT) (DELETE-UNO (CAR IT) L2)))
              (PERM IT L2) (ORDERED IT))
         (PERM (INSERT L1 IT) (CONS L1 L2))).
```

But simplification reduces this to T, using the `:definition PERM`.

Subgoal \*1.1/1

```
(IMPLIES (AND (CONSP IT) (<= L1 (CAR IT))
              (PERM IT L2) (ORDERED IT))
         (PERM (INSERT L1 IT) (CONS L1 L2))).
```

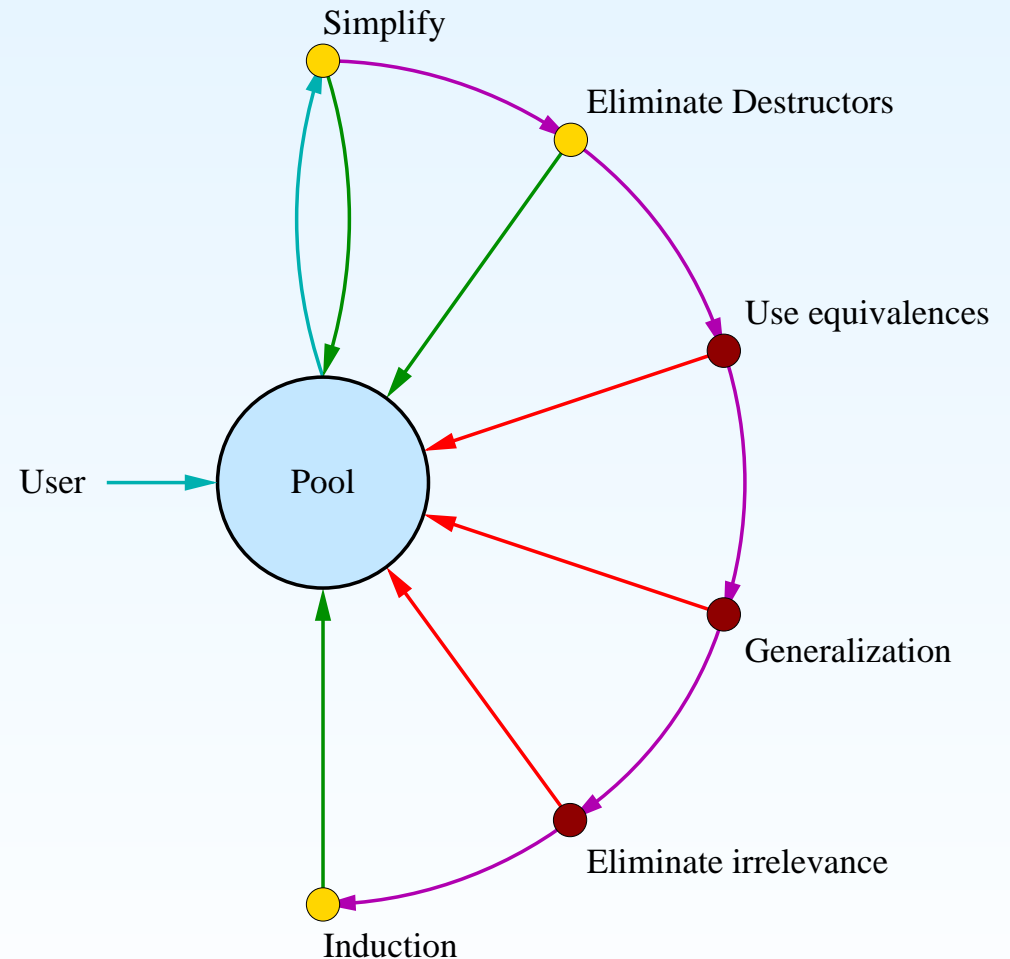
But simplification reduces this to T, using the `:definitions DELETE-ONE, INSERT, MEMBER, ORDERED` and `PERM`, primitive type reasoning, the `:rewrite` rules `CAR-CONS` and `CDR-CONS` and the `:type-prescription` rules `MEMBER` and `PERM`.

That completes the proofs of \*1.1 and \*1.

Q.E.D.

# The ACL2 theorem prover

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



## 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

---

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

```
(defun divide (p d mode)
  .....
  ;;; hundreds of lines faithfully
  ;;; describing 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 classical language for implementing symbolic computation systems
- The price to pay: a quantifier-free first-order logic
  - Limited expressiveness for stating properties
- A major case study: “*Formal Verification of Buchberger’s Algorithm*”, PhD Thesis, Inmaculada Medina Bulo

## A naive Common Lisp implementation of Buchberger algorithm

---

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

(defun Buchberger-aux (F C)
  (if (and (k-polynomials F)
          (k-polynomials-pairsp C))
      (if (endp C)
          F
          (let* ((p (first (first C)))
                 (q (second (first C)))
                 (h (red* (s-poly p q) F)))
            (if (equal h (|0|))
                (Buchberger-aux F (rest C))
                (Buchberger-aux (cons h F)
                                (append (pairs h F)
                                        (rest C)))))))
      F))
```

## The main theorem proved

---

```
(defthm |Buchberger(F) = <F>|
  (implies (and (k-polynomialp p)
                (k-polynomialsp F))
            (iff (in<> p F)
                  (equal (red* p (Buchberger F))
                          (|0|))))))
```

where:

- **(k-polynomialp p)**:  $p$  rational polynomial in  $k$  variables
- **(k-polynomialsp F)**:  $F$  finite set of polynomials in  $k$  variables
- **(in<> p F)**:  $p \in \langle F \rangle$
- **(red\* p (Buchberger F))**: normal form of  $p$  with respect to  $Buchberger(F)$
- **(|0|)**: The zero polynomial

## A sketch of the ACL2 proof

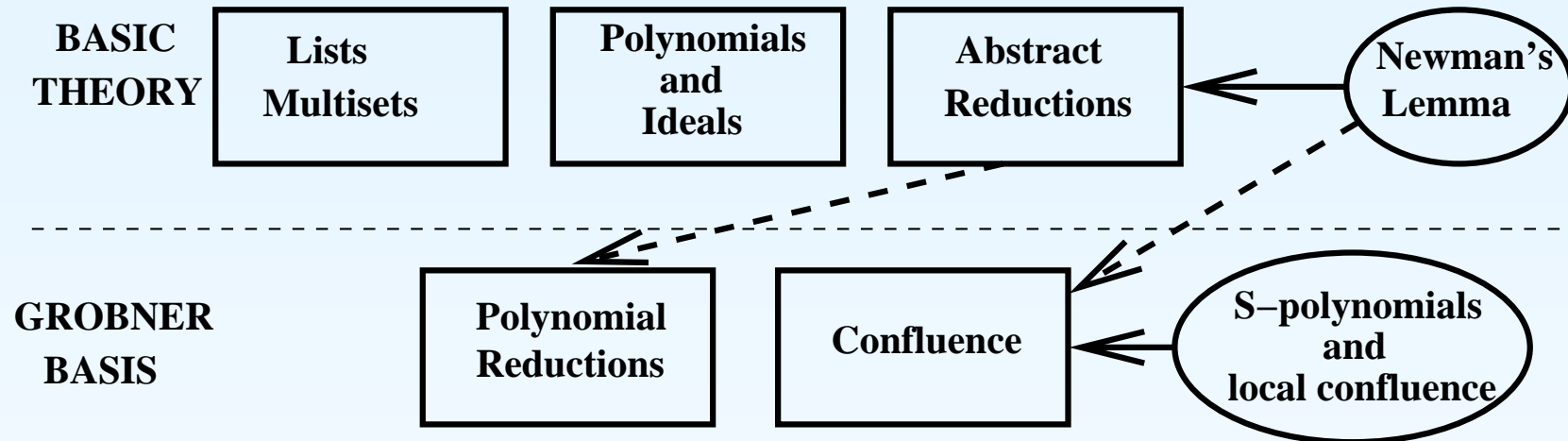
---

# A sketch of the ACL2 proof

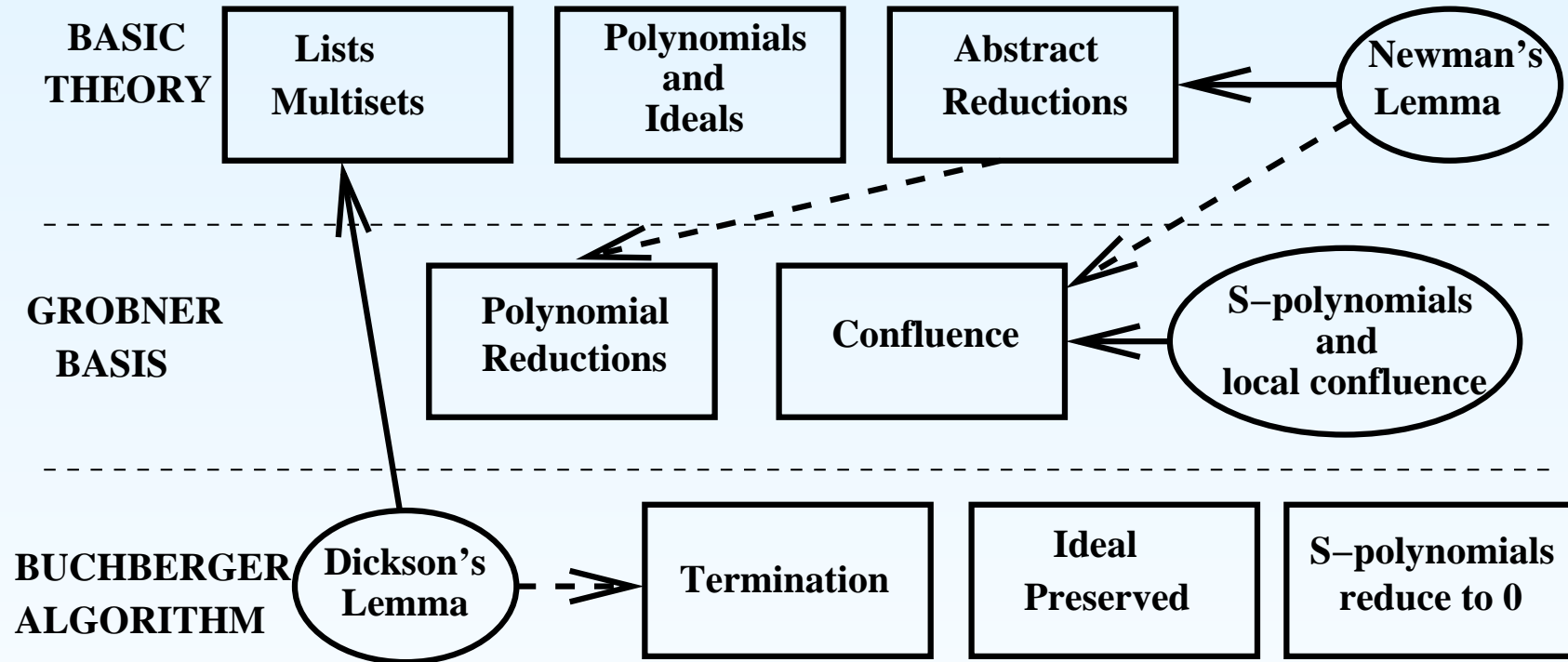
---



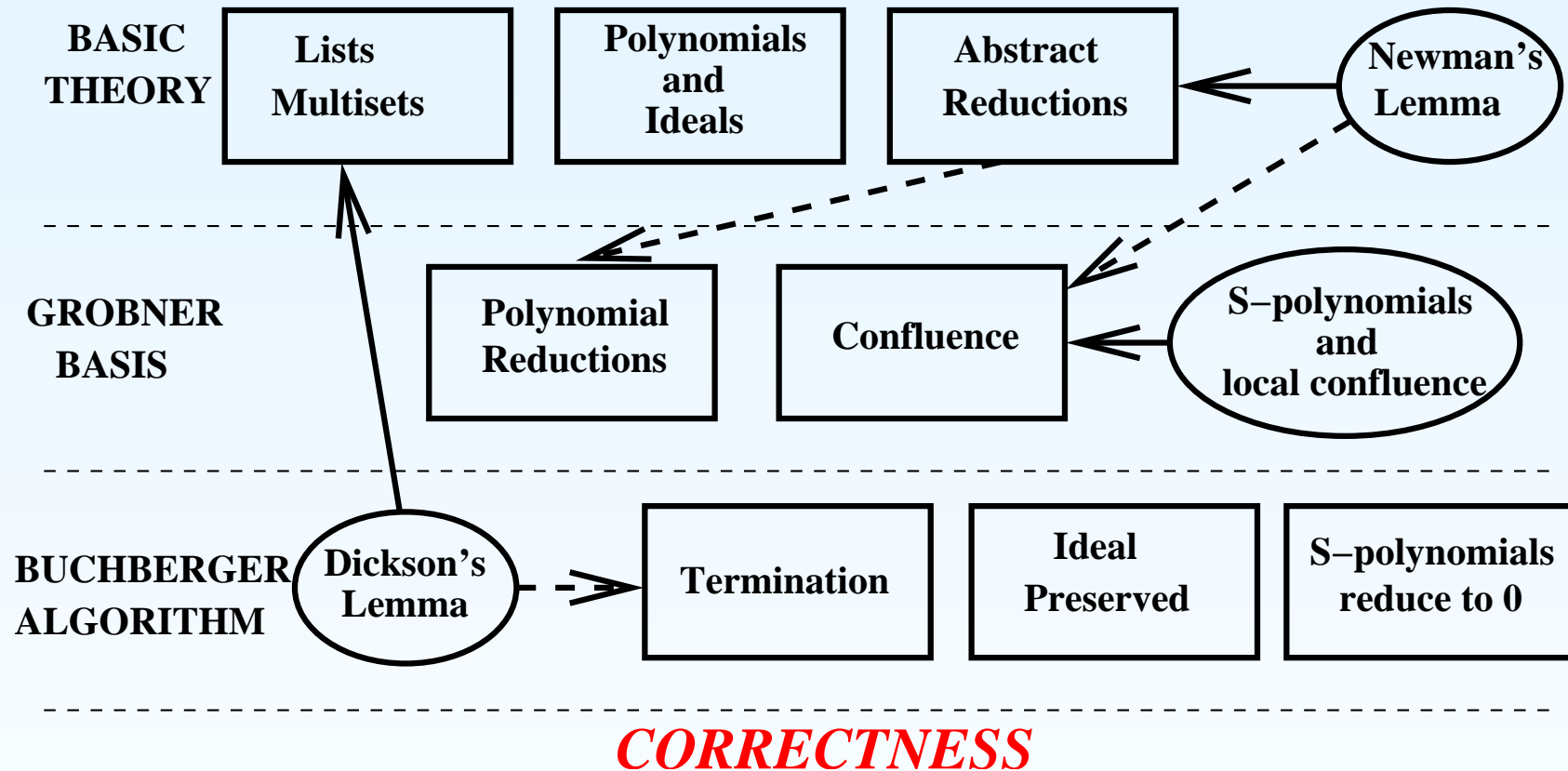
# A sketch of the ACL2 proof



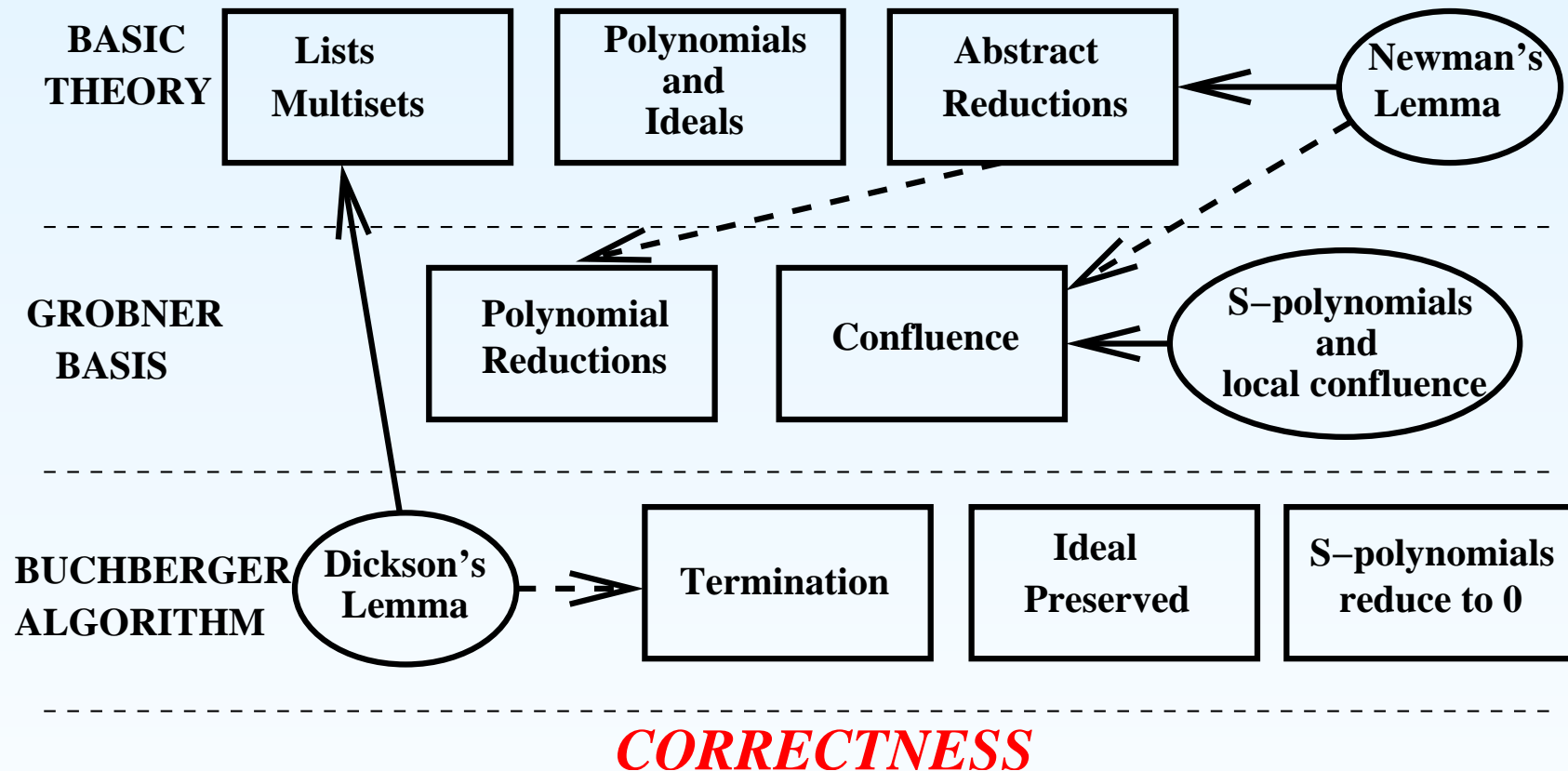
# A sketch of the ACL2 proof



# A sketch of the ACL2 proof



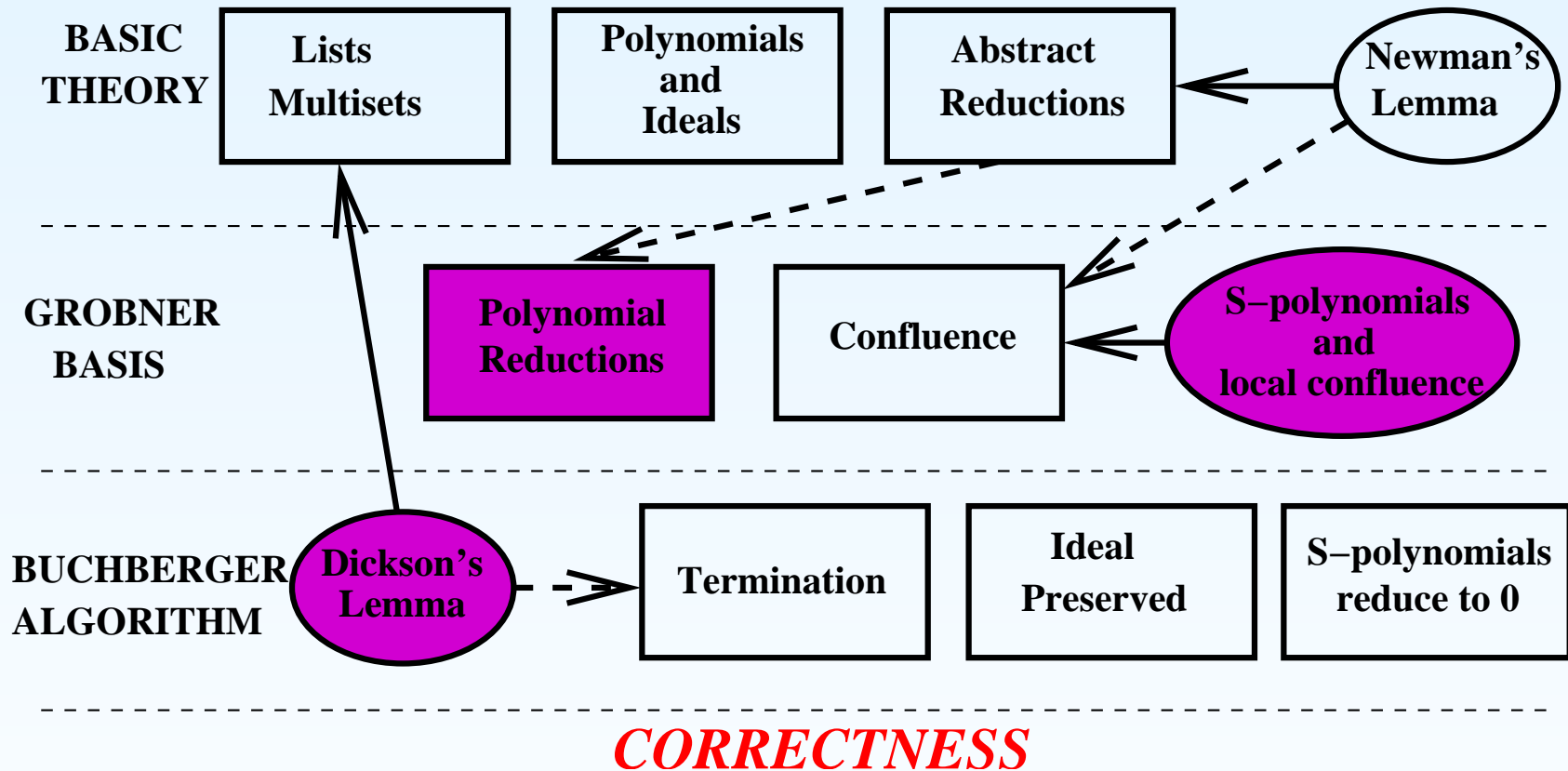
# A sketch of the ACL2 proof



The challenge:

- Formalize this mathematical theory in the ACL2 logic
- Lead the prover to this proof

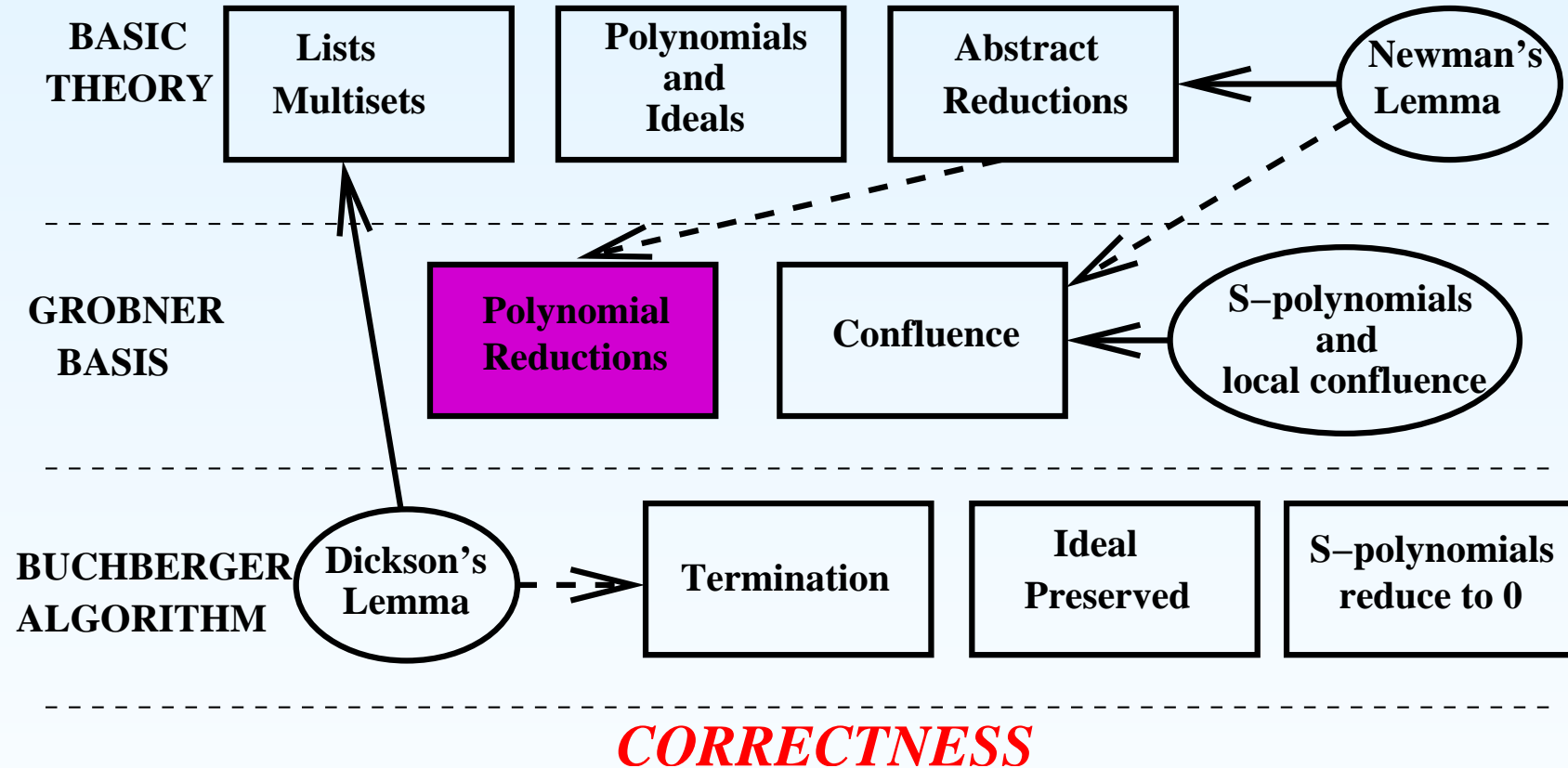
# A sketch of the ACL2 proof



The challenge:

- Formalize this mathematical theory in the ACL2 logic
- Lead the prover to this proof

# The polynomial reduction relation



## The polynomial reduction relation

---

- $p \rightarrow_F q$  if  $\exists f \in F, m \neq 0$  monomial in  $p$ , and  $c$  monomial, such that  $m = -c \cdot \text{lm}(f)$  and  $q = p + c \cdot f$ 
  - Problem: existential quantification in this definition
  - We have to explicitly deal with  $f, m$  and  $c$
  - *Operator*: a three element list  $(m, c, f)$
- We can define  $\rightarrow_F$  by means of three ACL2 functions:
  - `(k-polynomialp x)`: defines the domain of the reduction relation
  - `(reduction p o)`: applies an operator  $o$  to a polynomial  $p$
  - `(validp p o F)`: check if its *valid* apply  $o$  to  $p$ 
    - $f \in F, m$  monomial in  $p, \text{lm}(f)$  divides to  $m$  and  $c = -m/\text{lm}(f)$

## Equivalence closure

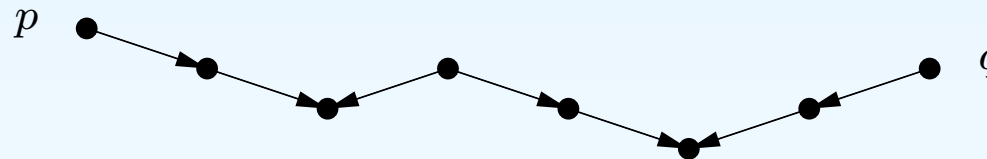
---

- We call  $p \leftrightarrow_F q$  a *proof step*
- It is represented by a (Lisp) structure with four fields, containing  $p$ ,  $q$ , the operator applied and the direction of the step.
- A proof step is *valid* if one of its two polynomials is obtained applying the (valid) operator to the other, in the indicated direction

```
(defun <-> (p q step F)
  (and (valid-proof-stepp step F)
        (equal p (elt1 step))
        (equal q (elt2 step))))
```

## Equivalence closure

- A *proof* is a list of concatenated valid proof steps
- Proofs allow us to constructively define the relation  $p \xleftrightarrow{*}_F q$



```
(defun <->* (p q proof F)
  (let ((r (elt2 (first proof))))
    (and (k-polynomialp p)
         (if (endp proof)
             (equal p q)
             (and (<-> p r (first proof) F)
                  (<->* r q (rest proof) F))))))
```

- $\rightarrow^*$  is similar to  $\leftarrow^*$ , but with all its steps from left to right



## A key theorem: local confluence and s-polynomials

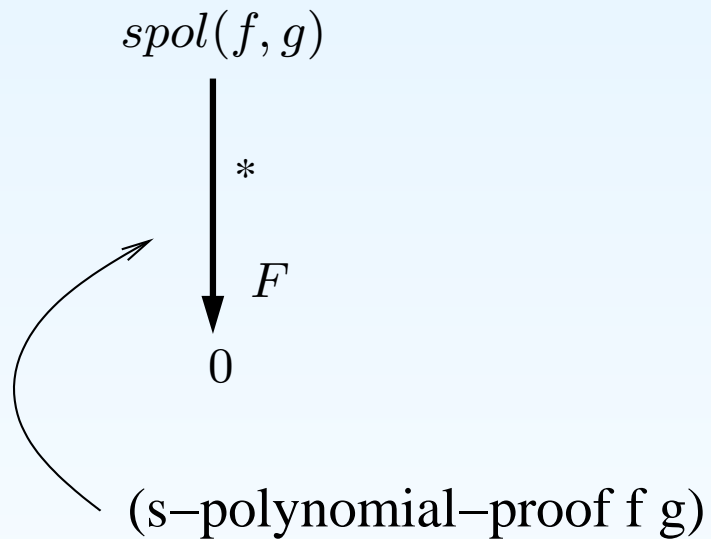
---

- $\rightarrow$  is locally confluent if  $\forall p, q, r$  such that  $p \leftarrow r \rightarrow q$  (*local peak*),  $\exists s$  such that  $p \xrightarrow{*} s \xleftarrow{*} q$  (*valley*)
- $\Phi(F) \equiv \forall p, q \in F \quad \text{spoly}(p, q) \xrightarrow{*}_F 0$
- Theorem: If  $\Phi(F)$  then  $\rightarrow_F$  locally confluent

# The key theorem viewed as a “proof shape” transformation

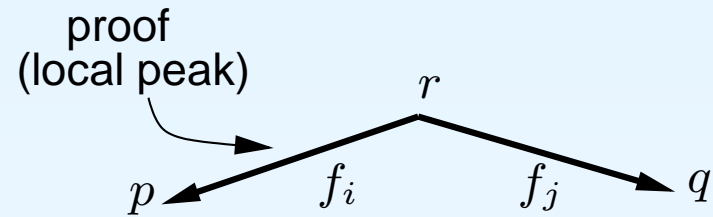
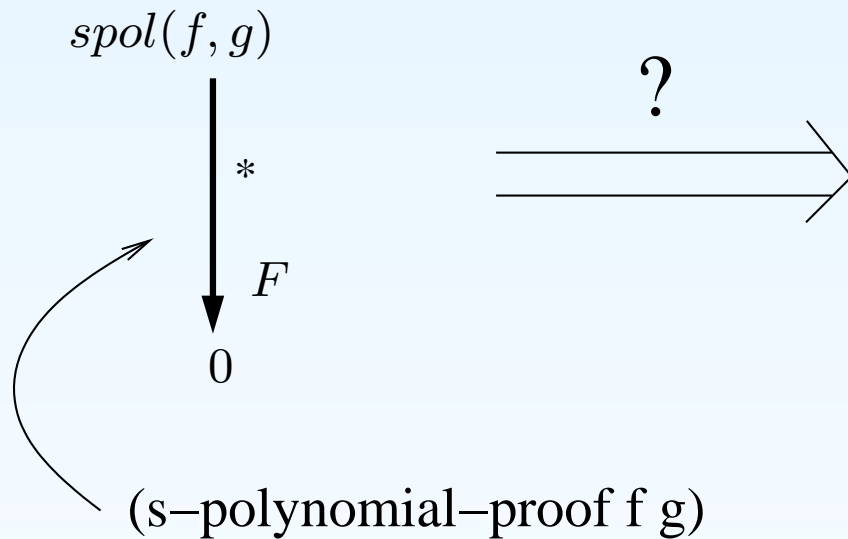
---

Assumption:  $\forall f, g \in F$



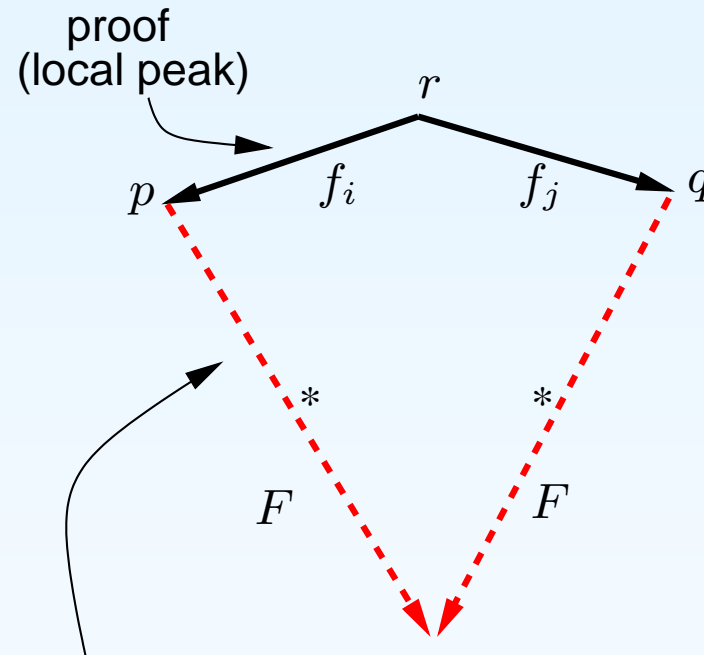
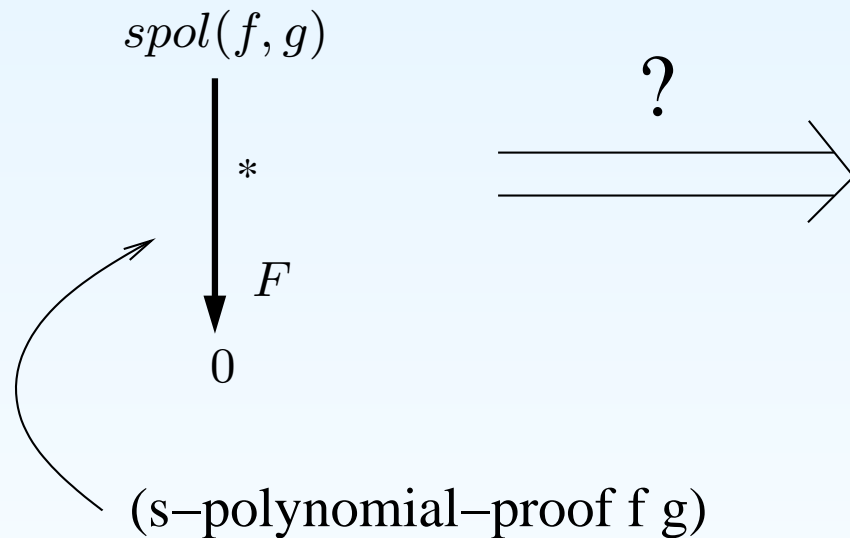
# The key theorem viewed as a “proof shape” transformation

Assumption:  $\forall f, g \in F$



# The key theorem viewed as a “proof shape” transformation

Assumption:  $\forall f, g \in F$



Goal: to define a function  
(transform-local-peak-F proof)  
obtaining an equivalent valley  
for every local peak proof  
(and prove its properties)

## Local confluence and s-polynomials: ACL2 formalization

---

Assumption:

```
(encapsulate
  ((F () t)
   (s-polynomial-proof (p q) t))
  .....
  (defthm |k-polynomialsp(F)|
    (k-polynomialsp (F)))

  (defthm |Phi(F)|
    (let ((proof (s-polynomial-proof p q)))
      (implies (and (in p (F)) (in q (F)))
                (->* (s-poly p q) (|0|) proof (F))))))
```

## Local confluence and s-polynomials: ACL2 formalization

---

### Conclusion:

```
(defthm |Phi(F) => local-confluence(->F)|  
  (let ((valley (transform-local-peak-F proof)))  
    (implies (and (polynomialp p) (polynomialp q)  
                  (local-peakp proof)  
                  (<->* p q proof (F)))  
              (and (<->* p q valley (F))  
                    (valleyp valley))))))
```

## Local confluence and s-polynomials

---

Three cases, depending on the monomials used for both reductions

```
(defun transform-local-peak-F (proof)
  (let ((m1 (o-monomial (operator (first proof))))
        (m2 (o-monomial (operator (second proof)))))
    (cond ((equal (term m1) (term m2))
           (transform-local-peak-F-= proof))
          ((RAC-TER::< (term m1) (term m2))
           (transform-local-peak-F-< proof))
          (t
           (transform-local-peak-F-> proof)))))
```

## Local confluence and s-polynomials

- **Lemma 1:** Let  $m$  be a non-zero monomial,  $p, q \in Q[X]$  and  $F \subseteq Q[X]$ . Then:

$$p \rightarrow_F^* q \implies m \cdot p \rightarrow_F^* m \cdot q$$

- **Lemma 2:** Let  $p, q \in Q[X]$  and  $F \subseteq Q[X]$ . Then:

$$p - q \rightarrow_F^* 0 \implies p \downarrow_F^* q$$

In ACL2, functions performing *proof transformations* :

```
(defun proof-|p ->F* q => m * p ->F* m * q|  
  (m proof)  
  ...)
```

```
(defun proof-|p - q ->*F 0 => p ->*F<- q|  
  (p q proof)  
  ...)
```

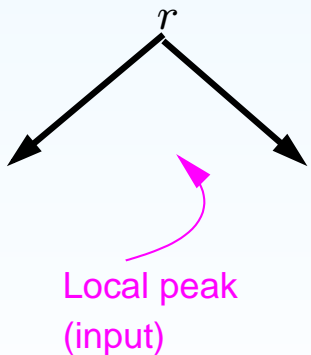
## Gröbner basis: critical overlap

---

```
(defun transform-local-peak-F-= (proof)
  (let* ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof)))))
    (proof-|p - q ->*F 0 => p ->*F<- q|
      (elt1 (first proof)) (elt2 (second proof))
      (proof-|p ->F* q => m * p ->F* m * q|
        (coeff-lcm m fj fi)
        (s-polynomial-proof fj fi))))))
```

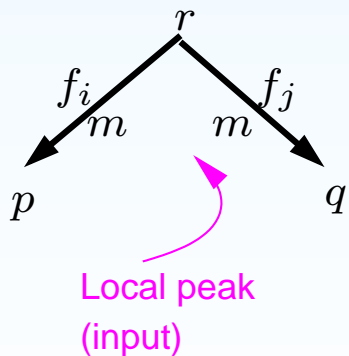
## Gröbner basis: critical overlap

```
(defun transform-local-peak-F-= (proof)
  (let* ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof))))
        (proof-|p - q ->*F 0 => p ->*F<- q|
          (elt1 (first proof)) (elt2 (second proof))
          (proof-|p ->F* q => m * p ->F* m * q|
            (coeff-lcm m fj fi)
            (s-polynomial-proof fj fi))))))
```



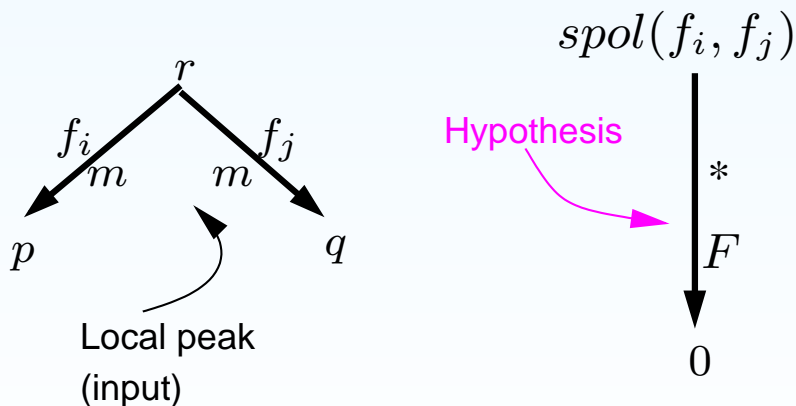
## Gröbner basis: critical overlap

```
(defun transform-local-peak-F= (proof)
  (let* ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof))))
        (proof-|p - q ->*F 0 => p ->*F<- q|
          (elt1 (first proof)) (elt2 (second proof))
          (proof-|p ->F* q => m * p ->F* m * q|
            (coeff-lcm m fj fi)
            (s-polynomial-proof fj fi))))))
```



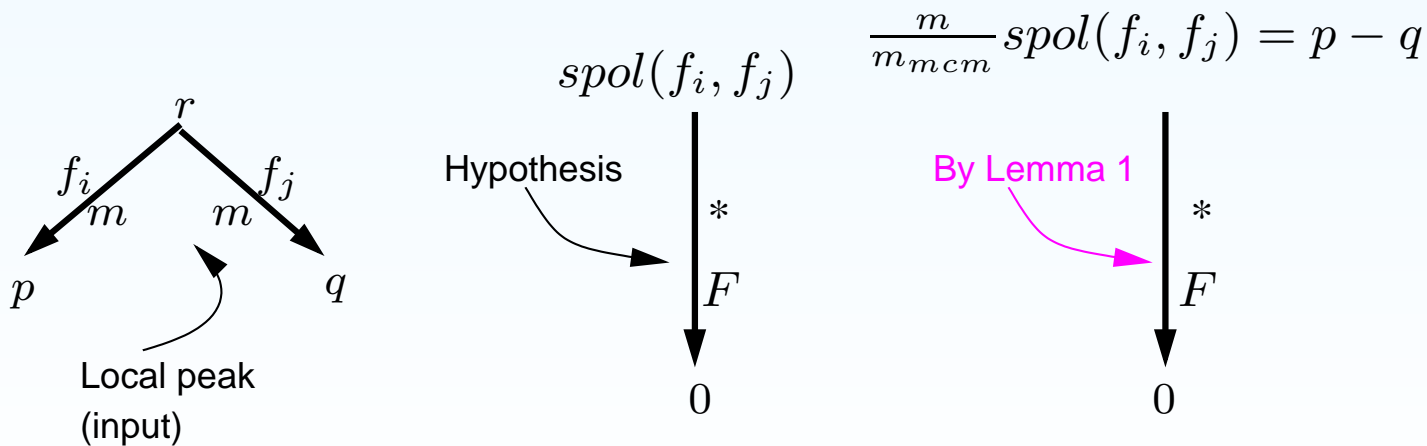
# Gröbner basis: critical overlap

```
(defun transform-local-peak-F-= (proof)
  (let* ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof))))
        (proof-|p - q ->*F 0 => p ->*F<- q|
          (elt1 (first proof)) (elt2 (second proof))
          (proof-|p ->F* q => m * p ->F* m * q|
            (coeff-lcm m fj fi)
            (s-polynomial-proof fj fi))))))
```



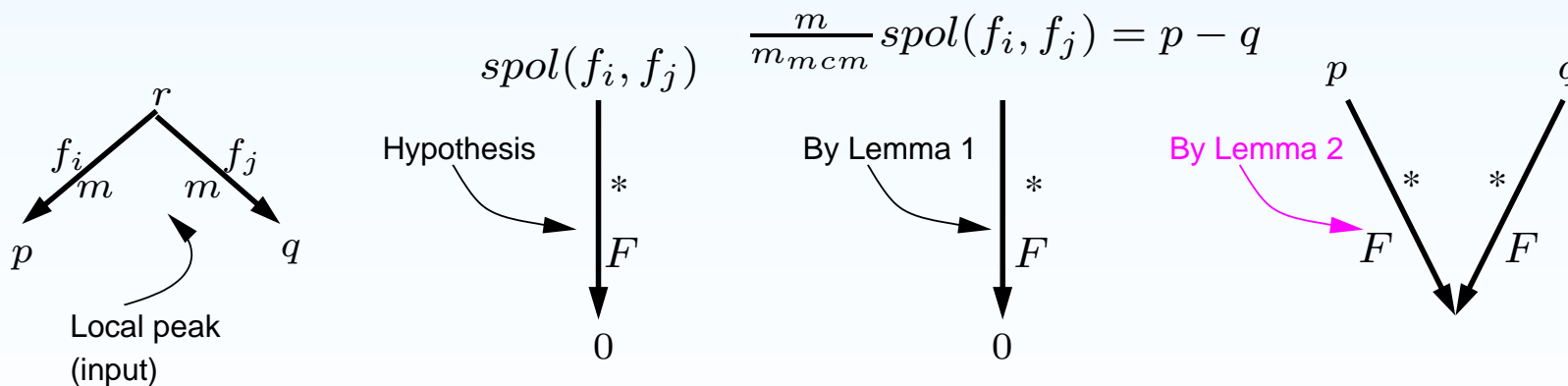
# Gröbner basis: critical overlap

```
(defun transform-local-peak-F-= (proof)
  (let* ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof))))
        (proof-|p - q ->*F 0 => p ->*F<- q|
          (elt1 (first proof)) (elt2 (second proof))
          (proof-|p ->F* q => m * p ->F* m * q|
            (coeff-lcm m fj fi)
            (s-polynomial-proof fj fi))))))
```

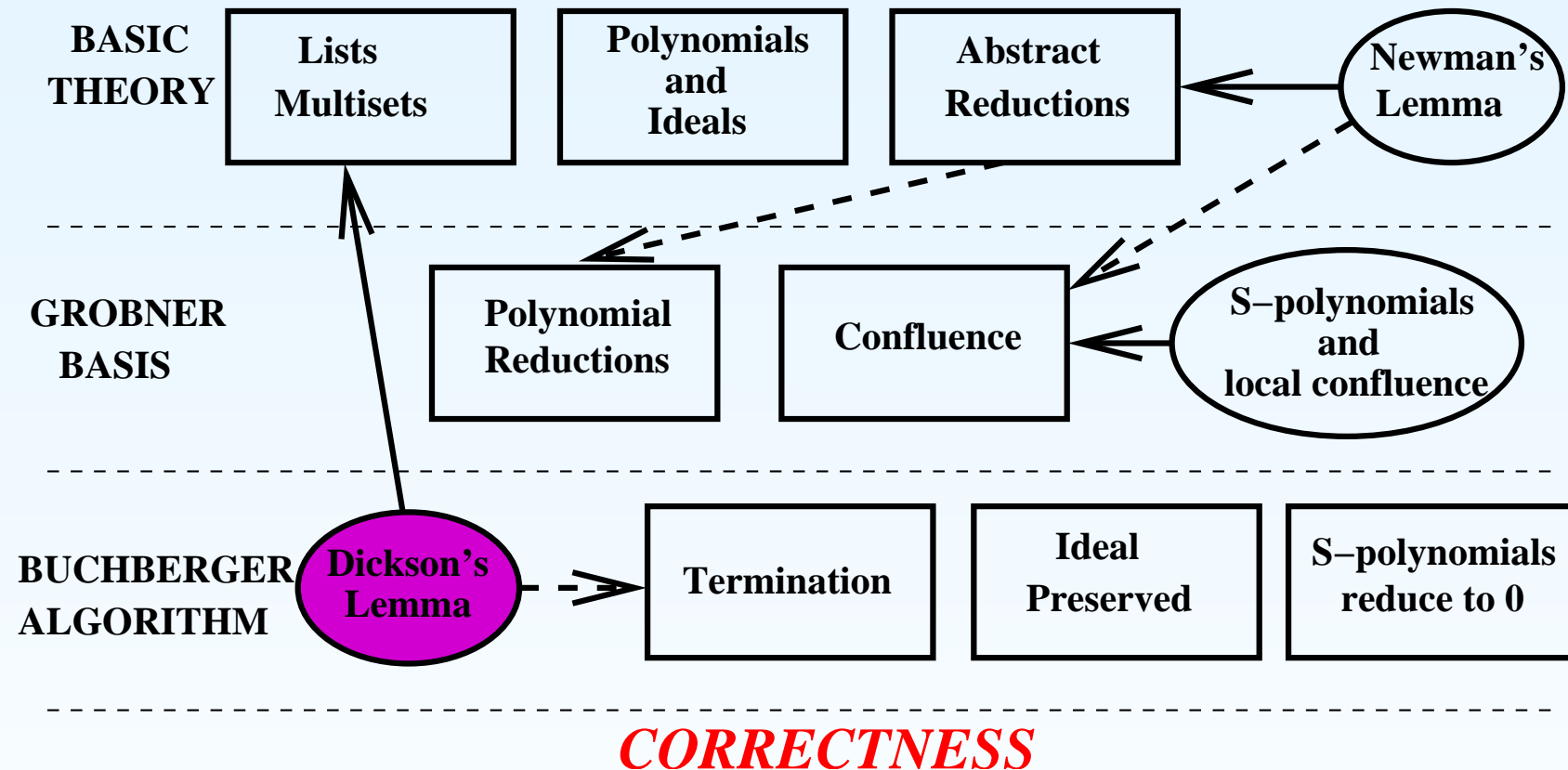


# Gröbner basis: critical overlap

```
(defun transform-local-peak-F-= (proof)
  (let* ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof))))
        (proof-|p - q ->*F 0 => p ->*F<- q|
          (elt1 (first proof)) (elt2 (second proof))
          (proof-|p ->F* q => m * p ->F* m * q|
            (coeff-lcm m fj fi)
            (s-polynomial-proof fj fi))))))
```



# A formal proof of Dickson's Lemma in ACL2



## A formal proof of Dickson's Lemma in ACL2

---

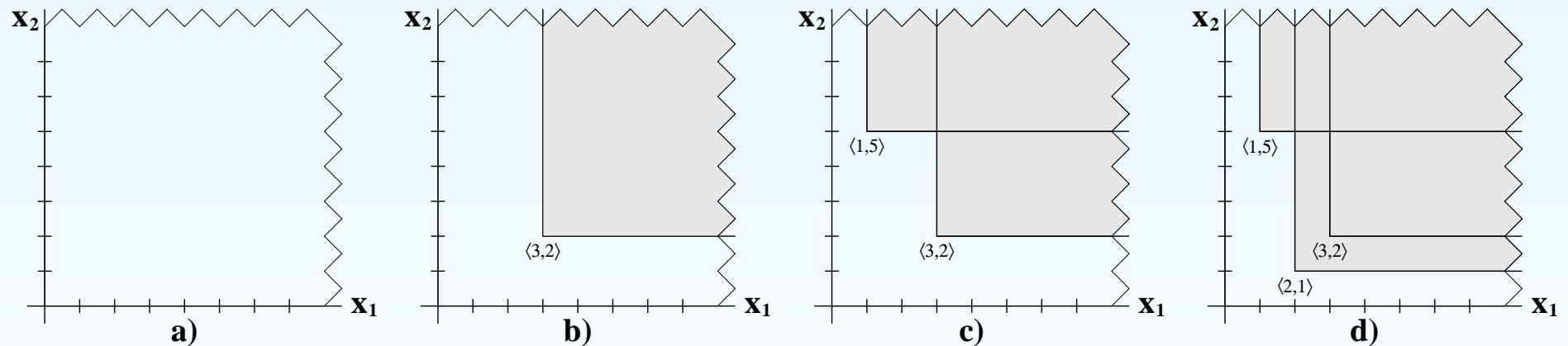
- Dickson's Lemma is the main result needed to prove termination of **Buchberger-aux**:

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

- Formalization in ACL2 is challenging:
  - Classical proofs are non-constructive
  - Absence of existential quantifiers:  $i$  and  $j$  has to be explicitly given

# A formal proof of Dickson's Lemma in ACL2

- Example: assume that  $m_0 = x^3y^2$ ,  $m_1 = xy^5$  and  $m_2 = x^2y$ .  
Graphically:



- The “free space” (non-shaded region) represents the set of “allowable” tuples at position  $k$ 
  - This free space decreases in a well-founded way

## A formal proof of Dickson's Lemma in ACL2

---

- Sketch of the proof carried out in ACL2:
  - Define a function assigning an ordinal (below  $\epsilon_0$ ) to every initial segment of the sequence, measuring (in some sense) the “free space”
  - Show that this ordinal decreases whenever the next tuple is not divisible by any of the previous tuples
- See details in <http://www.cs.us.es/~fmartin/acl2/dickson/> or in “*A Formal Proof of Dickson's Lemma in ACL2*”, F.J. Martín et al. LPAR-2003, LNCS 2850

## 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
  - Verifying real Common Lisp code
  - A certain degree of automation
- A formal proof of a mathematical result is interesting in its own
  - Detail, rigor and clarity
  - Deeper understanding
- An interesting by-product: the underlying mathematical theory of Buchberger's algorithm can be stated and proved in a quantifier-free logic

## 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
- And that was even taking into account that 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

## (Long term) Future work

- Verification of programs with more efficient data structures

## (Long term) Future work

- Verification of programs with more efficient data structures
- An alternative to program verification: output verification

## (Long term) Future work

- Verification of programs with more efficient data structures
- An alternative to program verification: output verification
- Sharing mathematical knowledge

## (Long term) 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

## (Long term) 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

## (Long term) 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
  - *Mathematical Knowledge Management*