Годишник на секция “Информатика”
Съюз на учените в България
Том 2, 2009, 14-24

Annual of “Informatics” Section
Union of Scientists in Bulgaria
Volume 2, 2009, 14-24

Verification Environment for Imperative and Object-Oriented Programs

Magdalina Todorova
Sofia University “St. Kliment Ohridski”, Faculty of Mathematics and Informatics
magda@fmi.uni-sofia.bg

Abstract: The paper presents a verification environment for imperative and object-oriented programs (using Hoare logic) in the frame of the HOL theorem prover. The environment is composed of a Hoare expression generator and verification language. An example of verification of OOP using the realized environment is proposed.

1. Introduction

The HOL theorem prover offers means for working with higher-order predicate logic formulae. The paper aims to enhance the HOL system with a practical verification environment for procedural and object-oriented programs. The environment is composed of a Hoare Expression Generator (HEG) and Verification Language (VL).

HEG takes program with pre- and postcondition {P} Program {Q}, additionally annotated with assertions, loop and class invariants and produces a Hoare-expression on the VL-language.

VL is a procedure language with superstructured axiomatic semantics. It is built in HOL theorem prover by deep embedding. The axiomatic semantics of the language is realized used Hoare inference rules. The proof of trueness of the Hoare expression corresponding to {P} Program {Q} via VL, verifies Program regarding P and Q.

2. Hoare Expression Generator

HEG is a translator, which translates a program in a procedure or object-oriented language, annotated adequately with precondition, postcondition and invariants in loops and classes into Hoare expression of the VL language. Some of the symbols for annotation of procedure and object-oriented programs used in this paper are:

Precondition P	P is a precondition
Postcondition Q	Q is a postcondition
Class_invariant CI	CI is a class invariant
Loop_invariant I	I is a loop invariant
Limited_function t	t is a loop limiting function
##…##	text between ## and ## gives preconditions, postconditions, invariants and limiting functions

For example HEG translates the procedure in C++ language

```
## Precondition    dividend >= 0 && divisor > 0;
   Postcondition  dividend == quot*divisor + rem && 0 <= rem &&
                  rem < divisor
##
void div_mod(int dividend, int divisor, int& quot, int& rem)
{ quot = 0;
  rem = dividend;
  while (divisor <= rem)
  { ## Loop_invariant dividend == quot*divisor + rem &&
                      0 <= rem && divisor > 0;
       Limited_function rem
    ##
    quot = quot + 1;
     rem = rem - divisor;
   }
 }
```

annotated with precondition, postcondition, invariant and limiting function of the loop and returning the quotient *quot* and remainder *rem* of the integer numbers division of *dividend* of *divisor* into the following Hoare expression

```
HOR ((%"dividend" GE #0) AND (%"divisor" GR #0))
     (procedure "div_mod"(val "dividend", val "divisor", ref "quot", ref "rem")
        (("quot" :== # 0);;
         ("rem" :== %"dividend");;
         (while ((% "dividend" EQ % "quot" MUL % "divisor" ADD % "rem")
                 AND (#0 LE % "rem") AND (% "divisor" GR #0))
               (% "rem")
               (% "divisor" LE % "rem")
                   (("quot" :== % "quot" ADD # 1);;
                ("rem" :== % "rem" SUB % "divisor")))))
        ((% "dividend" EQ % "quot" MUL % "divisor" ADD % "rem") AND
         (# 0 LE % "rem") AND (% "rem" LS % "divisor"))
```

in the VL language. This expression is a predicate. If it is valid the procedure div_mod is total correct regarding the pointed input/output specification. It could be considered as a theorem, but actually this theorem cannot be proved by any of known systems for theorem proving, including those of HOL. The last could become possible if HOL theorem prover was superstructured by a language, allowing to interpret an expression of the above type as a theorem. The VL language performs this function.

3. Programs verification language VL

The language is a procedural one with axiomatic superstructured semantics. Basic types of data of VL are Integer and Boolean. Its programs are sequences of procedures and functions composed by the following statements: empty (skip), assignment, sequence, conditional, for loop, for addressing a procedure or function. For the sake of brevity of this paper, a subset of the language is described.

Grammar

```
arith_expr ::=
            variable
           | integer
           | arith_expr POW arith_expr
           | arith_expr MUL arith_expr
           | arith_expr QUO arith_expr
           | arith_expr REM arith_expr
           | arith_expr ADD arith_expr
           | arith_expr SUB arith_expr
bool_expr ::=
             arith_expr EQ arith_expr
           |  arith_expr LS arith_expr
           |  arith_expr LE arith_expr
           |  arith_expr GR arith_expr
           |  arith_expr GE arith_expr
           |  bool_expr IMP bool_expr
           |  NOT bool_expr
           |  bool_expr AND bool_expr
           |  bool_expr OR bool_expr
command ::=
            skip
           | variable :== arith_expr
           | command ;; command
           | iff bool_expr command command
           | while bool_expr arith_expr bool_expr command
           | procedure string form_param command
           | function string form_param command
specification ::=
             {bool_expr} command {bool_expr}
      |  [bool_expr]
```

where POW, MUL, QUO, REM, ADD and SUB are arithmetic operators for raising to power, multiplication, integer number division, remainder of integer number division, addition and subtraction, IMP means implication, NOT – negation, AND and OR – conjunction and disjunction respectively, whereas comparison operators are: EQ (for equality), LS (for less), LE (for less or equal), GR (for greater), GE (for greater or equal). Command defines syntaxes of statements: *skip* is the empty statemen; *variable :== arith_expr* is the assignment statement; *command ;; command* is the sequence statement; *iff bool_expr command command* is the conditional statement; *while bool_expr arith_expr bool_expr command* is the loop statement, whereas the first two expressions are the invariant and the limiting function and the subsequent *bool_expr* and *command* are the condition and the body of the loop; *procedure string form_param command* defines a procedure, where string and form_param assign the name and the formal parameters, command defines the body of the procedure and *function string form_param command* defines a function, where string and form_param assign the name and formal parameters, command defines the body of the function. The formal parameters of procedures and functions are defined in the following way

```
form_param ::=
        val string
           | ref string
           | form_param , form_param
```

where formal parameter preceded by val is value parameter, and a formal parameter preceded by ref, a parameter variable.

{P} S {Q} is the triad of Hoare, [bool_expr] is the value of bool_expr.

Language logics

Besides the generally accepted procedure semantics, VL logic includes the logic of Hoare, described by the following inference rules:

The Skip Rule

$$\frac{\texttt{[P IMP Q]}}{\texttt{\{P\} skip \{Q\}}}$$

The Assignment Rule

$$\frac{\texttt{[P IMP Q(E} \leftarrow \texttt{x)]}}{\texttt{\{P\} x := E \{Q\}}}$$

The Sequence Rule

$$\frac{\texttt{\{P\} S1 \{R\}}\ \text{и}\ \texttt{\{R\} S2 \{Q\}}}{\texttt{\{P\} S1 ;; S2 \{Q\}}}$$

The Conditional Rule

$$\frac{\begin{array}{l}\texttt{\{P AND B\} S1 \{Q\}}\ \text{и}\\ \texttt{\{P AND (NOT B)\} S2 \{Q\}}\end{array}}{\texttt{\{P\}(iff B S1 S2)\{Q\}}}$$

Procedure Definition Rule

$$\frac{\texttt{\{P\} S1 \{Q\}}}{\texttt{\{P\}(procedure N R S1)\{Q\}}}$$

Function Definition Rule

$$\frac{\texttt{\{P\} S1 \{Q\}}}{\texttt{\{P\}(function N R S1)\{Q\}}}$$

The Loop Rule

$$\frac{\begin{array}{l}\texttt{[P IMP I]}\ \text{и}\\ \texttt{\{I AND B\} S \{I\}}\ \text{и}\\ \texttt{[(I AND (NOT B)) IMP Q]}\ \text{и}\\ \texttt{[(I AND B) IMP (t GR 0)]}\ \text{и}\\ \texttt{[(I AND B) IMP Wp(t1 := t ;; S, t LS t1)]}\end{array}}{\texttt{\{P\}(while I t B S)\{Q\}}}$$

where P, Q, R and B are Boolean expressions, E is an arithmetic expression, S, S1 and S2 are arbitrary statements of the VL language, x and t1 are variables, t is a loop limiting function. Q(E←x) is a Boolean expression, in which every appearance of the variable x is replaced by the expression E. The loop rule contains an invariant I and a limiting function t of the loop. The first 3 prerequisites express partial correctness, whereas the other two – the termination of the loop statement. One of the prerequisites of the conclusion rule of the loop statement contains the transformation predicate Wp(command, bool_expr) [1], defined in the following way:

```
Wp(skip, Q) = Q
Wp(x :== E, Q) = Q[E←x]
Wp(S1;; S2, Q) = Wp(S1, Wp(S2, Q))
Wp(iff B S1 S2, Q) = (B IMP Wp(S1, Q)) ∧
                     (NOT B IMP Wp(S2, Q))
```

If S is a program of the VL language, the verification of S regarding P and Q is reduced to proving the trueness of the Hoare triple {P} S {Q}. For the purpose, the described inference rules are applied.

Realization

Implemented by building in the syntax and semantics of the VL language in the HOL theorem prover by deep embedding. The Kananaskis 4 of HOL [2] version was used. A subset of the implementation is annexed at the end of the paper.

Building in the syntax

Implemented by the means of HOL for definition of data types. Inbuilt are the definitions of integer arithmetic expressions (using type arith_expr), Boolean expressions (using type bool_expr), formal parameters (using type form_param) and statements (using type command).

The integer variables of VL are strings and their values are designated by string preceded by the % symbol. For example "a" and "index" are names of integer variables and % "a" and % "index" are their values. Integer constants are designated by integer numbers preceded by the # symbol. For example # 2 and # -5912 are two integer constants of the VL language. Arithmetic operators are infixed. Their priorities are as those of procedure languages. Higher priority number assigns an operation of higher priority. Using

```
set_fixity "POW" (Infixr 99);
```

a priority number 99 is assigned to the infixed right associative arithmetic operator for raising to power POW. The remaining arithmetic operators are left associative. Infixl assigns priorities numbers and their associative ness.

Comparison operators are infixed ones; left associative with equal priority numbers and compare arithmetic expressions only. NOT denial is a prefix operator and conjunction and disjunction are infixed and left associative.

Operators ":==", ";;" and "," are infixed and right associative.

Example. Function

```
int exp(int base, int exponent)
{ int i = exponent;
  int product = 1;
  while (i >= 1)
   { product = product * base;
     i = i – 1;
   }
   return product;
}
```

returning $base^{exponent}$ (exponent is a given integer number, exponent ≥ 1), in the VL language is recorded in the following way:

```
function "exp"(val "base", val "exponent")
 ( "i" :== % "exponent" ;;
   "product" :== #1 ;;
   (while
      ((# 0 LE % "i") AND (% "i" LE % "exponent") AND
       (%"product" EQ % "base" POW (% "exponent" SUB % "i")))
       (%"i")
       (# 1 LE % "i")
       ("product" :== % "product" MUL % "base" ;;
        "i" :== % "i" SUB #1)) ;;
   "return" :== % "product")
```

Function has two formal parameter values and four statements – three for assignment and a loop statement. Statement while contains:

loop invariant I:	((# 0 LE % "i") AND (% "i" LE % "exponent") AND (%"product" EQ % "base" POW (% "exponent" SUB % "i")))
limiting function t:	% "i"
loop condition:	# 1 LE % "i"
loop body:	"product" :== % "product" MUL % "base" ;; "i" :== % "i" SUB #1

Building in the semantics

Realized by building in the semantics of the arithmetic and Boolean expressions, statements and inference rules. For the purpose are used the HOL means for functions definition. Building in of procedure and axiomatic semantics in realization are independent of each other. The annex at the end of the paper includes the realization of the axiomatic semantics only. Building in of Integer and Boolean expressions is realized using functions Se and Sb, respectively:

Se : arith_expr → state → num
Sb : bool_expr → state → {true, false}

The second parameter of these functions is the *state*, at which the expression is calculated. Function state is of string ->num type.

Building in of inferace rules is realized using function:

HOR : bool_expr → command → bool_expr → {true, false}

The definition of the HOR function uses the transforming predicate Wp:

Wp : command → bool_expr → bool_expr

Implementation of Wp for the assignment statement is linked to the implementation of a substitution of each appearance of the variable with an arithmetic expression in a Boolean expression. The latter is done by the addressing (SUBST_bool Q E V), by means of which every appearance of V in the Boolean expression Q is substituted by the arithmetic expression E. For Boolean expressions, which are comparisons to arithmetic expressions, substitution is reduced to replacement of each appearance of a variable with an expression in an arithmetic expression. The addressing (SUBST_arith A E V) to the function SUBST_arith realizes substitution of every appearance of the variable V with the expression E in the arithmetic expression A.

Let us get back to the power raising function of the example. In order to verify *exp* regarding precondition *P: exponent ≥ 1* and postcondition *Q: product = base*^{exponent}, we should assess the Hoare expression:

```
HOR (# 1 LE % "exponent")
      (function "exp"(val "base", val "exponent")
          ("i" :== % "exponent";;
           "product" :== #1 ;;
            (while
                ((# 0 LE % "i") AND (% "i" LE % "exponent") AND
                  (%"product" EQ % "base" POW (% "exponent" SUB % "i")))
                (%"i")
                (# 1 LE % "i")
                ("product" :== % "product" MUL % "base" ;;
                 "i" :== % "i" SUB #1)) ;;
            "return" :== %"product"))
       (% "product" EQ % "base" POW % "exponent")
```

using the HOL system, into which the realization of the VL is built in. After building in the VL, the above Hoare expression becomes a HOL theorem.

4. An example of verification of OOP using the realized environment

Object-oriented programs are sets of classes. Let P_i and Q_i be the precondition and postcondition for i^{th} constructor C_i or method M_i of a class, whereas the invariant of the class, to which these methods pertain, is a Class_invariant. In order to verify a class regarding relevant assertions, it is necessary to prove the trueness of the Hoare triples:

{P_i} C_i {$Q_i \wedge$ Class_invariant}, for each constructor C_i

and

{$P_i \wedge$ Class_invariant} M_i {$Q_i \wedge$ Class_invariant}, for each method M_i of the class [3].

For verification of a class all constructors and member functions of the class are annotated according to the syntax of the HEG translator using adequate preconditions, postconditions, loop and classes invariants. HEG translates the annotated class into a theorem equal to conjunction of the Hoare expressions, corresponding to each constructor and member function.

The next class implements the return of the quotient and remainder of the integer number division. The class comprises a constructor and member function for returning the quotient *quot* and the remainder *rem* of the integer number division of member data *divident* and *divisor*.

```
## Class_invariant divisor != 0
##
class C
{ public:
      ## Precondition first_val >= 0 && second_val > 0
          Postcondition dividend >= 0 && divisor > 0
      ##
      C(int first_val, int second_val)
      { dividend = first_val;
        divisor = second_val;
      }
      ## Precondition dividend >= 0 && divisor > 0;
          Postcondition dividend == quot*divisor + rem &&
                            0 <= rem && rem < divisor
      ##
      void div_mod(int& quot, int& rem)
      { quot = 0;
        rem = dividend;
        while (divisor <= rem)
        { ## Loop_invariant dividend == quot*divisor + rem &&
                                   0 <= rem && divisor > 0;
              Limited_function rem
          ##
          quot = quot + 1;
          rem = rem - divisor;
         }
       }
    private:
       int dividend, divisor;
};
```

HEG translates the annotated class into conjunction of the Hoare expressions, corresponding to the constructor and the member function div_mod.

```
HOR ((%"first_val" GE #0) AND (%"second_val" GR #0))
        (procedure "C"(ref "dividend", ref "divisor",
                              val "first_val", val "second_val")
         ("dividend" :== %"first_val";;
          "divisor"  :== %"second_val"))
         ((%"dividend" GE #0) AND (%"divisor" GR #0) AND
           (NOT (#0 EQ %"divisor"))) /\
HOR ((%"dividend" GE #0) AND (%"divisor" GR #0) AND
         (NOT (#0 EQ %"divisor")))
        (procedure "div_mod"(val "dividend", val "divisor",
                                       ref "quot", ref "rem")
             ("quot" :== # 0 ;;
          "rem” :== %"dividend" ;;
      while ((%"dividend" EQ %"quot" MUL %"divisor" ADD %"rem")
                   AND (#0 LE % "rem") AND (#0 LS % "divisor"))
                 (% "rem")
                 (% "divisor" LE % "rem")
                 ("quot" :== % "quot" ADD # 1 ;;
                  "rem" :== % "rem" SUB % "divisor")))
                ((%"dividend" EQ %"quot" MUL %"divisor" ADD %"rem")
                  AND (#0 LE % "rem") AND (% "rem" LS % "divisor")  AND
                   (NOT (#0 EQ %”divisor”)))
```

Hoare expressions are theorems of the HOL language. The prove of the above theorem using the HOL command

```
-e (ARW_TAC[HOR_def, Wp_def, Sb_def, SUBST_bool, Se_def, SUBST_arith]);
```

verifies the class with regard to the indicated specifications.

5. Conclusion

In recent years researches related to the creation of verification systems of procedure and object-oriented programs have been initiated. Experiments with the described environment for verification of procedure and simple object-oriented programs show good prospects. The next task is widening the environment by adding data types and statements, relevant to modern object-oriented languages.

Acknowledgements

The paper is supported by Grant 173/2009 from Sofia University Research Fund.

References

[1] D. Gries, The Science of Programming, Springer-Verlag, Berlin and New York, 1981.
[2] http://hol.sourceforge.net/ HOL 4, Kananaskis 4, 2007.
[3] http://www.cs.wm.edu/~coppit/other-papers/tucker_noonan_ch12.pdf, 2005.

```
val ARW_TAC = RW_TAC arith_ss;

open arithmeticTheory;
load "stringLib";
load "numLib";

Hol_datatype `arith_expr =
  % of string
| # of num
| ADD of arith_expr => arith_expr
| SUB of arith_expr => arith_expr
| POW of arith_expr => arith_expr
| MUL of arith_expr => arith_expr
| QUO of arith_expr => arith_expr
| REM of arith_expr => arith_expr`;

set_fixity "POW" (Infixr 99);
set_fixity "MUL" (Infixl 97);
set_fixity "QUO" (Infixl 97);
set_fixity "REM" (Infixl 97);
set_fixity "ADD" (Infixl 95);
set_fixity "SUB" (Infixl 95);

val bool_expr =
     Hol_datatype `bool_expr =
            EQ  of arith_expr => arith_expr
           | LS  of arith_expr => arith_expr
           | LE  of arith_expr => arith_expr
           | GR  of arith_expr => arith_expr
           | GE  of arith_expr => arith_expr
           | IMP of bool_expr => bool_expr
           | NOT  of bool_expr
           | AND  of bool_expr => bool_expr
           | OR  of bool_expr => bool_expr`;

val form_param =
     Hol_datatype `form_param =
            val of string
           | ref of string
           | , of form_param => form_param`;

Hol_datatype `command =
 skip
| :== of string => arith_expr
| ;;    of command => command
| iff     of bool_expr => command => command
| while of bool_expr => arith_expr => bool_expr =>
command
| procedure of string => form_param => command
| function of string => form_param => command`;

set_fixity "IMP" (Infixl 90);
set_fixity "EQ" (Infixl 90);
set_fixity "LS" (Infixl 90);
set_fixity "LE" (Infixl 90);
set_fixity "GR" (Infixl 90);
set_fixity "GE" (Infixl 90);
set_fixity "AND" (Infixl 83);
set_fixity "OR" (Infixl 83);
set_fixity ":==" (Infixl 20);
set_fixity ";;" (Infixr 10);
set_fixity "," (Infixr 10);

val Se_def = Define
`(Se (% v) s = s v) /\
  (Se (# c) s = c) /\
  (Se (m POW n) s = ((Se m s) ** (Se n s))) /\
  (Se (m MUL n) s = ((Se m s) * (Se n s))) /\
  (Se (m QUO n) s = ((Se m s) DIV (Se n s))) /\
  (Se (m REM n) s = ((Se m s) MOD (Se n s))) /\
  (Se (m ADD n) s = ((Se m s) + (Se n s)) ) /\
  (Se (m SUB n) s = ((Se m s) - (Se n s)))`;

val Sb_def = Define
`(Sb (m EQ n) s = ((Se m s) = (Se n s))) /\
 (Sb (m LS n) s = ((Se m s) < (Se n s))) /\
 (Sb (m LE n) s = ((Se m s) <= (Se n s))) /\
 (Sb (m GR n) s = ((Se m s) > (Se n s))) /\
 (Sb (m GE n) s = ((Se m s) >= (Se n s))) /\
 (Sb (a IMP b) s = ((Sb a s) ==> (Sb b s))) /\
        (Sb (NOT a) s = ~(Sb a s)) /\
 (Sb (a AND b) s = ((Sb a s) /\ (Sb b s))) /\
 (Sb (a OR b) s = ((Sb a s) \/ (Sb b s)))`;

val SUBST_arith = Define
`(SUBST_arith (% p) E V =
      (if p = V then E else (%p))) /\
 (SUBST_arith (#c) E V = (#c)) /\
 (SUBST_arith (m POW n) E V =
      ((SUBST_arith m E V) POW
        (SUBST_arith n E V))) /\
 (SUBST_arith (m MUL n) E V =
      ((SUBST_arith m E V) MUL
        (SUBST_arith n E V))) /\
 (SUBST_arith (m QUO n) E V =
      ((SUBST_arith m E V) QUO
        (SUBST_arith n E V))) /\
 (SUBST_arith (m REM n) E V =
      ((SUBST_arith m E V) REM
        (SUBST_arith n E V))) /\
 (SUBST_arith (m ADD n) E V =
      ((SUBST_arith m E V) ADD
        (SUBST_arith n E V))) /\
 (SUBST_arith (m SUB n) E V =
      ((SUBST_arith m E V) SUB
        (SUBST_arith n E V)))`;

val SUBST_bool = Define
`(SUBST_bool (m EQ n) X V =
       ((SUBST_arith m X V) EQ
         (SUBST_arith n X V))) /\
  (SUBST_bool (m LS n) X V =
       ((SUBST_arith m X V) LS
         (SUBST_arith n X V))) /\
  (SUBST_bool (m LE n) X V =
       ((SUBST_arith m X V) LE
        (SUBST_arith n X V))) /\
  (SUBST_bool (m GR n) X V =
        ((SUBST_arith m X V) GR
         (SUBST_arith n X V))) /\
   (SUBST_bool (m GE n) X V =
     ((SUBST_arith m X V) GE (SUBST_arith n X V))) /\
   (SUBST_bool (a IMP b)X V =
((SUBST_bool a X V) IMP (SUBST_bool b X V))) /\
```

```
  (SUBST_bool (NOT a) X V =   NOT(SUBST_bool a X V)) /\
  (SUBST_bool (a AND b) X V =
       ((SUBST_bool a X V) AND (SUBST_bool b X V))) /\
   (SUBST_bool (a OR b) X V =
        ((SUBST_bool a X V) OR (SUBST_bool b X V)))`;

val Wp_def = Define
` (Wp skip Q = Q) /\
   (Wp (V :== E) Q = (SUBST_bool Q E V)) /\
   (Wp (S1 ;; S2) Q = (Wp S1 (Wp S2 Q))) /\
    (Wp (iff B S1 S2) Q =  ((B IMP (Wp S1 Q))
               AND (NOT B IMP (Wp S2 Q)))) /\
   (Wp (while B E B1 S1) Q = B)`;

val HOR_def = Define
 `(HOR P skip Q = (!s. (Sb P s) ==> (Sb Q s))) /\
 (HOR P (V :== E) Q =
   (!s. (Sb P s) ==>
       (Sb (Wp (V :== E) Q) s))) /\
 (HOR P (S1 ;; S2) Q = ((HOR P S1 (Wp S2 Q)) /\
                                 (HOR (Wp S2 Q) S2 Q))) /\
   (HOR P (iff B S1 S2) Q = (!s. (Sb P s) ==>
                              (Sb (Wp (iff B S1 S2) Q) s))) /\
   (HOR P (while B1 E B2 S1) Q =
        (!s. ((Sb P s) ==> (Sb B1 s)) /\
        ((Sb B1 s) /\ (Sb (NOT B2) s) ==> (Sb Q s)) /\
        (HOR (B1 AND B2) S1 B1) /\
        ((Sb B1 s) /\ (Sb B2 s) ==> (((Se E s) > 0) /\
        (Sb (Wp (("t1" :== E ) ;; S1)
              (E LS % "t1")) s) )))) /\
  (HOR P (procedure N R S1) Q = (HOR P S1 Q)) /\
  (HOR P (function N R S1) Q = (HOR P S1 Q))`;

val Lemma1 = store_thm
("Lemma1", ``!x. ~(1 <= x) ==>
         (x = 0)``, ARW_TAC[]);
val Lemma2 = store_thm
("Lemma2",  ``x  *  x  **  k  =  x  **  (k  +  1)``,
ARW_TAC[EXP_ADD]);
```