A *miniKanren* Interactive Proof System for Deciding the Truth of (simple) Quantified Boolean Formulae

Larisse D. Voufo

April 24th, 2007

Abstract

We present an application of *miniKanren* with respect to the needs of Proof and Complexity Theories. We implement an Interactive Proof System for deciding the truth of quantified boolean formulae, which completes the proof that the complexity classes IP(class of problems solvable by an interactive proof system) and PSPACE(class of decision problems requiring polynomial space, but may need exponential time) are equal. We experiment on the reversibility capabilities of *miniKanren* and discover a potential to extend our implementation to all variants of interactive proof systems.

Contents

1 Introduction: Motivations **2**

2 QBF and the Proof of PSPACE $\subseteq$ IP **3**
2.1 Overview on Quantified Boolean Formulae 3
2.2 An Interactive Proof System for Deciding the Truth of QBF 6
2.2.1 Additional remarks: 7

3 Implementation: *Scheme* **7**
3.1 Utility tools 8
3.2 One-way library functions 8
3.3 IPSys 8
3.4 Limitations: prime numbers generation is finite 9

4 Implementation: *miniKanren* **10**
4.1 More Utility tools 10
4.2 Reversible library functions 10
4.3 mk_IPSys 10

5 Experiments and Analysis **12**

6 Future work **15**

7 Conclusion **15**

A Overview on Interactive Proof Systems **16**
A.1 Brief History 17
A.2 Definition: 17
A.3 Variants 17
A.3.1 MIP: 17
A.3.2 IPP: Unbounded IP 18
A.3.3 compIP: competitive IP 18

B The Proof of IP=PSPACE **18**

C Simulating an Interactive Proof System by a PSPACE machine **19**

D The *mk_ arithmetic* module **22**

E Where is the "*schemish*" folder? **26**

1 Introduction: Motivations

Interactive Proof Systems (IPS) For a long time, the question of how a verifier can be convinced with high probability that a given theorem is provable without showing the whole proof, and of how rapidly this can be done, remained an open problem. This led interested parties to formulate and extensively study it in terms of "*interactive protocols*" .

In 1992, Adi Shamir surprisingly completed the proof of IP = PSPACE, allowing IP to be placed in the standard classification of feasible computations[2]. This proof amazingly showed that when both randomization and interaction are allowed, the proofs that can be verified in polynomial time are exactly those proofs that can be generated with polynomial space[1].

To do that, Adi Shamir illustrated the existence of an IPS for solving the PSPACE-complete problem of deciding the truth of Quantified Boolean Formulae (QBF) [1] (cf. 2.2). Notice that the understanding of this illustration then majorly relies on that of QBF and some operations on them.

Outline In this paper, we think it would be interesting to observe an implementation of the IPS defined in the illustration in question. Furthermore, we would like to extend that implementation to all possible variants (Appendix A.3) of IPS[2], by simply modifying or adding a prover, a verifier, or the channels. Thus, we present a first implementation of the IPS using a functional programming language called *Scheme*[6, 8]. This sets us up for (the implementation of) a good portion of currently existing interactive protocols. Then, in order to account for all possible variants, including those that rely heavily on reversibility (such as Quantum IPS), we also experiment with implementing it in *miniKanren.*

[1] To remain within the scope of this paper, information about the proof of IP=PSPACE (and IPS in general) has been added to the appendices.

[2] Discussed in the appendices.

MiniKanren (mK) *MiniKanren* could be viewed as a relational programming language implemented in *Scheme. It is a Scheme* implementation, of a declarative logic programming system, recently developed by Daniel Friedman, William Byrd, and Oleg Kiselyov, and used in their Book "The Reasoned Schemer"[9]. A major interesting component of this language is its support for reversible programming, which we will be experimenting here.

This said, this paper will consist of three(3) main parts, first we will present Adi Shamir's IPS for deciding the truth of QBF(2). Then, we will present our *Scheme*(3), followed by our *miniKanren*(4), implementations of that IPS. We will present elegant tools that had to be put in place in order to bring these implementations to life, the actual implementations in question, and the results of experiments and analysis of those results[3].

As a side note, all the codes used in this experiment, and referred to in the rest of this paper are gathered in a folder called "*schemish*" (cf. E). Please refer to its content for additional implementation details.

2 QBF and the Proof of PSPACE ⊆ IP

The proof for $PSPACE \subseteq IP$ is credited to Adi Shamir[1]. In his "$IP = PSPACE$" paper, he presents some concise notes on QBF and $PSPACE \subseteq IP$. Here, in order to maintain the conciseness of his statements, we will replicate some of those notes. However, we will also extend them to be better presented and to include more clarifications whenever needed. Meanwhile, citations from his paper will simply be double-quoted (the implicit reference being [1]).

As stated earlier, a major part of that proof relies on understanding some operations on QBF as defined in Adi Shamir's "$IP = PSPACE$" paper[1]; hence, the next section.

2.1 Overview on Quantified Boolean Formulae

- ▷ "A **Quantified Boolean Formula** is the closure of the set of Boolean variables x, and their negations $\overline{x}$, under the operations $\wedge$(and), $\vee$(or), $\forall$(universal quantification), and $\exists$(existential quantification)".

  Under this scheme, the operations =(equality) and $\implies$ (implication) are translated into $(a \wedge b) \vee (\bar{a} \wedge \bar{b})$ for $a = b$, and $\bar{a} \vee b$ for $a \implies b$, given some variables a and b[1].

- ▷ "A **closed QBF** is a QBF in which all the variables are quantified. It can be evaluated to either T(true) or F(false)".

- ▷ "An **Open QBF with** $k > 0$ **free variables** can be interpreted as a boolean function from $\{T, F\}^k$ to $\{T, F\}$".

- ▷ "A **closed QBF** is called **simple** if in the given syntactic representation, every occurrence of each variable is separated from its point of quantification by at most one universal quantifier (and arbitrarily many other symbols)".

  Example:

[3]Consult the attached "schemish" folder for a full working implementation.

"$\forall x_1 \, \forall x_2 \, \exists x_3 \, [(x_1 \vee x_2) \wedge \forall x_4 \, (x_2 \wedge x_3 \wedge x_4)]$ is simple.

However, $\forall x_1 \, \forall x_2 \, [(x_1 \wedge x_2) \wedge \forall x_3 \, (\bar{x_1} \wedge x_3)]$ is not simple, due to an extra quantifier between $\forall x_1$ and the usage of x_1 in $(\bar{x_1} \wedge x_3)$".

▷ "Notice that every QBF of size n can be transformed into an equivalent simple QBF whose size is polynomial in n".

Example:

"$\exists x_1 \, \forall x_2 \, \exists x_3 \, \forall x_4 \, \exists x_5 \ldots Q(x_1, x_2, x_3, \ldots)]$, where Q is quantifier free, can be transformed into:

$\exists x_1^0 \, \forall x_2^0 \, \exists x_1^1 \, (x_1^1 = x_1^0) \wedge \exists x_3^0 \forall x_4^0 \, \exists x_1^2 \, \exists x_2^1 \, \exists x_3^1 \, (x_1^2 = x_1^1) \wedge (x_2^1 = x_2^0) \wedge (x_3^1 = x_3^0) \wedge \exists x_5^0 \ldots Q(x_1^{j_1}, x_2^{j_2}, x_3^{j_3}, \ldots)$,

where $x_i^{j_i}$ is the last name given to the initial Boolean variable x_i^0".

"This QBF is simple by definition, and contains a quadratic number of variables (compared to the original QBF), since each one of the original n variables is being renamed under a linear bound on n".

Arithmetizing QBF

▷ The **arithmetization of a given QBF** is its transformation into an arithmetic form, under the following scheme[1]:

- ▶ Boolean variable $x_i \longrightarrow$ new variable z_i,
- ▶ $\overline{x_i} \longrightarrow (_i)$
- ▶ $\wedge$, $\vee$, $\forall x$, $\exists x \longrightarrow *, +, \prod_{z_i \in \{0,1\}}, \sum_{z_i \in \{0,1\}}$; respectively.

Running example:

"Let $B = \forall x_1 \, [\bar{x_1} \vee \exists x_2 \, \forall x_3 \, (x_1 \wedge x_2) \vee x_3]$,

Its arithmetic form is: $A = \prod_{z_1 \in \{0,1\}} [(1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_2) + z_3]$, with value 2".

▷ Notice that "a closed QBF is henceforth true iff the value of its arithmetic form is nonzero".

▷ However, "this value can get quite large and has an upper bound of $O(2^{2^n})$".

"As an illustration, notice that for any (potentially open) subexpression B of a given QBF, its maximal value v, under all possible 0/1 substitutions to the free variables, satisfies the following (where v' and v'' are the maximal values of B' and B'' respectively):

- ▶ If B is x_i or $\overline{x_i}$, then $v = 1$.
- ▶ If B is $B' \vee B''$, then $v \leq v' + v''$.
- ▶ If B is $B' \wedge B''$, then $v \leq v' \cdot v''$.

- ▶ If B is $\exists x_i\, B'$, then $v \leq 2v'$.
- ▶ If B is $\forall x_i\, B'$, then $v \leq v'^2$.
- ▶ If B is closed, then v coincides with the value of its arithmetic form".

Thus, the worst case scenario happens when the given QBF is constituted of B, with the simplest operation between variables, nested within a succession of $\forall$ clauses as in the formula: $\forall x_1\, \forall x_2\, \forall x_3 \ldots \forall x_n\, B$.

Given that each universal quantifier squares the previous value, the maximal value of this formula is: $((((v^2)^2)^2)\ldots)^2 = v^{2\cdot 2\cdot 2\cdot \ldots \cdot 2} = v^{2^n}$.

Now, out of the remaining operations (excluding $\forall$), substituting v' and v'' by their maximal variable value of 1 results in $v = 2$; hence producing an upper bound of $O(2^{2^n})$.

▷ Since V cannot handle such large numbers, Modulo Arithmetic can be used to reduce them modulo some smaller prime p. The idea is that "given the arithmetic value A of a closed QBF B, there exists a prime p of length polynomial in the size of B, such that $A \neq 0 (mod\ p)$ iff B is true".

Functionalizing QBF Given a closed arithmetic expression A,

▷ "The **functional form** A', of A, is computed by eliminating the leftmost $\prod_{z_i \in \{0,1\}}$, or $\sum_{z_i \in \{0,1\}}$; and considering A' as a polynomial function $q(z_i)$ of one free variable z_i".

Running example:

"The functional form of A is: $A' = [(1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_2) + z_3]$.

By evaluating all the summations and products, P can express it as the polynomial $q(z_1) = z_1^2 + 1$".

▷ "The **randomized form** of A is $A'(z_i = r)$ in which z_i is set to a random number r modulo p supplied by the verifier".

"This randomized form can again be evaluated to a constant, but the number of $\prod$ and $\sum$ symbols is reduced by 1". Note that "the simplicity of A is preserved by these transformations".

Running example:

"The randomized form of $q(z_1)$ with $z_1 = 3$ is: $A'(z_1 = 3) = [(1-3) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (3 \cdot z_2) + z_3]$.

Its value 10 can be deduced from applying 3 to the function $q(z_1)$".

▷ Notice that "the degree of $q(z_1)$ can be exponentially high", due to products. However, "we can reduce it by simplifying the original expressions".

The idea is that "if a QBF is simple, then the degree of the polynomial $q(z_1)$ that describes the functional form of its arithmetic form grows at most linearly with the

size of the QBF". Also, "by adding sufficiently many dummy variables to the quantifier-free sub-expressions of the QBF, this degree can be reduced to 3, and thus allowing $q(z_1)$ to be represented by just four prime numbers".

2.2 An Interactive Proof System for Deciding the Truth of QBF

Given the arithmetic form A of a simple QBF B, We want to prove that $A \neq 0 \pmod p$ for some polynomially long prime p. For this purpose, there are three important steps:

1. **First step of the interactive proof**:

   "P chooses p and sends it to V, along with a written proof of primality, since even an infinitely powerful cheating prover cannot find such a prime if $A = 0$".

2. **During the next steps of the interaction protocol:**

   "V randomly chooses p, using the fact that for most p, $A \neq 0 \pmod p$ iff B is true".

   ▷ Notice that "one can prove both membership and non-membership by the same protocol".

   ▷ However, "if the chosen p happens to be a divisor of A, even an honest prover may be unable to prove a correct statement. Therefore, this protocol has imperfect completeness".

   Recall what we saw in part 2.1 about the exponentially high value of the degree of the polynomial $q(z_i)$ and how we can improve it by simplifying the original QBF.

3. **The very simple interactive protocol for proving that** $A \neq 0 \pmod p$.

   ▷ "P sends the claimed value a of $A \pmod p$ to V, and justifies this claim by considering successively smaller subexpressions of A".

   ▷ "At any intermediate stage of the protocol, the current expression A is split into $A_1 + A_2$ or $A_1 \cdot A_2$, where A_1 is a polynomial with fully instantiated variables (whose value a_1 can be computed by V himself), and A_2 starts with the leftmost Π or Σ symbol of A".

   ▷ "P and V then repeatedly execute the following simplification steps:

   (a) If A_2 is *empty*, V stops and accepts the claim iff $a = a_1$.

   (b) If A_1 is *nonempty*, V replaces A by A_2, and replaces a by $(a - a_1) \pmod p$ or $(a/a_1) \pmod p$ (depending on the operator that connects A_1 and A_2).
   If V tries to divide a by $a_1 = 0 \pmod p$, he stops and accepts the claim iff $a = 0 \pmod p$.

   ▷ Otherwise,

   ► P sends the polynomial descriptions $q(z_i)$ of A' to V.

   ► V checks that $a = (q(0)+q(l)) \pmod p$ or $a = (q(0) \cdot q(l)) \pmod p)$ (depending on the first symbol of A_2), sends a random $r \in Z_P$ to P, replaces A by $(A'(zi = r)) \pmod p$, and replaces a by $q(r) \pmod p$".

With this protocol, we notice two main things:

1. "When B is true and P is honest, V always accepts the proof".
2. "When B is false, V accepts the proof with negligible probability".

PROOF :

1. "An honest P can always justify his claimed values and polynomials".
2. "A cheating prover who supplied an incorrect value of a must provide an incorrect polynomial $q(zi)$ to support his claim, since a is checked against $q(0) + q(l)(\text{mod } p)$ or $q(0) \cdot q(l)(\text{mod } p)$. By the interpolation theorem, such an incorrect polynomial of degree t can agree with the correct polynomial on at most t of the p points in Z_P. When the value of t is a polynomial and the value of p is exponential in the size of B, there is only a negligible probability that the incorrect q yields a correct value when evaluated at a random point r chosen by V. As a result, a cheating P is forced to provide incorrect values for successively smaller sub-expressions, until he is exposed with overwhelming probability when V evaluates the final sub-expression by himself".

2.2.1 Additional remarks:

In his paper, Adi Shamir states that "a single application of this protocol suffices to make the probability of cheating exponentially small, and there is no need to iterate it as in other interactive proofs. However, the protocol seems to be inherently sequential, and it is a major open problem whether it can be executed with a small (e.g., logarithmic) number of rounds. Note that the existence of a constant round protocol for IP would collapse the polynomial hierarchy to its second level..."[1].

Weak Verifiers can improve the space-bound required by the protocol He also points out the fact that this interactive protocol requires polynomial time and polynomial space verifiers, and demonstrates a way to greatly improve the space bound using a **weak verifier**, which is characterized by the following:

- ▷ Its running-time is polynomial,
- ▷ Its workspace is logarithmic,
- ▷ It has a two-way read-only access to a random tape, and
- ▷ Its messages consist solely of the random bits it reads.

3 Implementation: *Scheme*

As stated earlier, the first step in our experiment is to implement Adi Shamir's IPS, as defined in part 2.2, in the *Scheme* programming language. We set in place various interesting library helpers for that purpose.

3.1 Utility tools

A major first step in this implementation involves gathering helper tools very efficiently. Here, two issues are worth noticing. First, the code duplication across multiple files (e.g. files loading/including) is an inefficient software design technique. Then, to take better advantage of *Scheme* compilers, the codes must be optimized the best and safest way possible. These issues are taken care of by defining a utilities library file called "utils.ss".

In this file, we parameterize all helper files location names to, of course, remove the need to modify several files each time a loaded or included file's location changes. Then, we import the "scheme" module for optimization purposes and separate our utilities functions into several modules to avoid unnecessary loadings of function definitions. This also serves as an efficient way to gather all of the useful functions into the minimal number of files possible. Furthermore, we make sure to take advantage of macro constructs whenever possible.

The *displayer*, *testing*, and *environment* modules elegantly defines some commonly used (and needed) functions and macros; the first one helping more for debugging purposes.

This utilities file also sets initializing "flags", for all newly created files, such as the case-sensitivity and other compiler options. From now on, each file must simply load "utils.ss" first thing in its creation and then make the best usage possible of its parameterized variables and of its modules, as demonstrated in the others files in the "schemish" folder (Appendix E).

3.2 One-way library functions

After creating the utility file, and as we are going one step further towards our IPS, we need to define the QBF operations described earlier in part 2.1. we do this in the file "qbf.ss"[4]. Notice that after loading "utils.ss", this file loads additional helper files using the parameterized variables for the names of the locations of those files. It also imports only the necessary modules; resulting in a performance boost [7].

We define the functions *qbf_sanitize*, *qbf_closed?*, *qbf_open?*, *qbf_simple?*, *qbf_simplify*, *qbf_san_simplify*, *qbf_arithmetize*, *qbf_evaluate**, *qbf_functional**, *qbf_apply**, *qbf_randomize**. These definitions account for both the general cases with values small enough to fit in a computer word (no need to pass in a prime number as parameter), and the cases where using modulo arithmetic is necessary[5].

The file also contains tests cases, for the above operations, all wrapped up within a single function call to *qbf_runtest*.

3.3 IPSys

Now, it is time for the actual IPS. But, we need to define a prime number generator beforehand. For the moment, let's use a prime numbers look up table and let's define it in a module in "*primes.ss". The module also defines helper functions for accessing elements in the list of prime numbers either in an orderly fashion or randomly. Our IPS is then constituted of the following prover and verifier defined in "IPsys.ss". For a given QBF, the system sanitizes and

[4]Here, all the function names are prefixed with "qbf_"

[5]using *case-lambda* clauses

simplifies it, translates it into its arithmetized form, calls the prover on that arithmetized form, and the chain of interactions begins.

```
;;; Prover
;;; 2 handling types:
;;; - first call from IPsys: need
;;;   to initialize some values
;;; - request, from V, for the
;;;   functional form of A, q
;;;
;;; Arguments:
;;;  A - arithmetic form of qbf
;;;  p - prime number
;;;  a - value of A
;;;
(define Prover
 (case-lambda
  [(A)
   (let ([err 10])
    (letrec (
     [prime
      (lambda ()
       (let* (
         [p (get-rand-prime-up err)]
         [a (if (eqv? p err)
              err
              (qbf_evaluate_arith A p))] )
        (cond
         [(eqv? p err) (values a p)]
         [(zero? a) (prime)]
         [else (values a p)]) ))] )
     (let-values ([(a p) (prime)])
       (if (eqv? p err)
         reject
         (Verifier A a p)))))]
  [(A p)
   (qbf_functional_arith A p)]))
```

```
;;; Verifier - 3 cases
(define Verifier
 (lambda (A a p)
  (match A
   [,a1 (guard (number? a1))
    (if (= a a1) accept reject)]
   [(,a1 + ,A2)
    (let ([a2 (modulo (- a a1) p)])
      (Verifier A2 a2 p))]
   [(,a1 * ,A2) (guard (zero? a1))
    (if (zero? a) accept reject)]
   [(,a1 * ,A2)
    (let ([a2 (modulo (/ a a1) p)])
      (Verifier A2 a2 p))]
   [,expr (guard
     (or (eqv? (car expr) 'sum)
         (eqv? (car expr) 'product)))
    (let* (
      [q (Prover A p)]
      [q0 (qbf_apply_func q 0)]
      [q1 (qbf_apply_func q 1)]
      [a-check
       (if (eqv? (car expr) 'sum)
         (modulo (+ q0 q1) p)
         (modulo (* q0 q1) p))] )
     (if (= a a-check)
       (let* (
         [r (get-rand-prime-up)]
         [A (qbf_randomize_arith A r p)]
         [a (modulo (qbf_apply_func q r) p)] )
        (Verifier A a p))
       reject) )]
    [,else (error 'Verifier
       "Invalid qbf: ~s\n" else)] )))
```

Table 1: Our *Scheme* implementation of IPS

3.4 Limitations: prime numbers generation is finite

As should be expected, the look-up table as means of prime numbers generation is not complete. The number of prime numbers in storage is finite; which means that running out of prime numbers in the course of the program's execution does not necessarily mean that we are in a rejecting state. However, we can assume that the prime numbers are large enough, and their number in storage is also sufficiently large to consider the relative error in

evaluation to be negligible.

4 Implementation: *miniKanren*

The second part of our experiment is the translation of our *Scheme* implementation of Adi Shamir's IPS into a *miniKanren* one. Again, we set in place additional interesting library functions for that purpose.

4.1 More Utility tools

We already started creating the utilities file ("utils.ss") earlier. Here, we extend it to include *miniKanren*-related helpers. we define the modules *displayer_mk*, *mk_arithmetic*, *mk_helpers*, and *mk_environment*. Out of all these, a very interesting one is *mk_arithmetic*.

An extension of *miniKanren*'s numerical system to include signed integers Currently, the numbers representation and arithmetic operations defined in mK only accounts for positive integers. This is not enough for our IPS's implementation. Therefore, we define the *mk_arithmetic* module to add support for signed integers.

Here, all numbers should be preceded by a positive ("pos") or negative ("neg") label for their signs. the number is constructed from a macro construct named "numl", which takes in a decimal number and builds its mK number representation[6], tagged with the appropriate label depending on its sign. We also extend the mK's arithmetic operations to "signed" ones, defining among others *addl*, *subl*, *mull*, and *divl*, for signed addition, subtraction, multiplication, and division respectively. Appendix D contains the module definition.

4.2 Reversible library functions

Now, we simply translate the implementation in "qbf.ss" into the mK language and save it into "mk_qbf.ss". Here, all function names are postfixed with "*_mk*".

In this translation, we realize that it is currently impossible to define an mK version of *qbf_simplify* because of scoping issues. The function "symbolize*"[7], which returns a new symbol given a variable name and an index by using the *Scheme* functions *string->symbol*, *string-append*, *symbol->string*, and *number->string*, strictly obeys static scoping rules, whereas an mK version of it would require dynamic scoping.

In addition, for easier parsing, all variables used in a given QBF are tagged with a "var" label. The utilities functions *label-vars* and *unlabel-vars*, from the *mk_helpers* module, are defined for easy transaction from a *Scheme*-based QBF to a mK-based one and vice-versa.

4.3 mk_IPSys

Again, we effectuate another translation from a *Scheme*-based "environment" to a mK one. Due to the restriction on the possibility to implement a QBF's simplification, we let the

[6] A reversed binary representation of the decimal number

[7] See the definition of *symbolyze2* in that of *qbf_simplify*.

responsibility to simplify a given QBF user-dependent. The following is the resulting "reversible" prover and verifier.

```
;;; Prover
;;; 2 handling types:
;;; - first call from IPsys: need
;;;   to initialize some values
;;; - request, from V, for the
;;;   functional form of A, q
;;;
;;; Arguments:
;;;  A - arithmetic form of qbf
;;;  p - prime number
;;;  a - value of A
;;;
(define Prover
 (case-lambda
  [(A outcome
  ; err should be a non-prime number
  (define err_numb 10)
  (define err (numl err_numb))
  (define prime
   (lambda (a p)
    (fresh (p0 a0)
    (== p0 (numl
      (get-rand-prime-up err_numb)))
    (conde
     [(== p0 err) (== p p0)
      (== a err)]
     [(never-equalo p0 err)
      (qbf_evaluate_arith_mk A p0 a0)
      (conde
       [(== a0 (numl 0)) (prime a p)]
       [(never-equalo a0 (numl 0))
        (== p p0) (== a a0)]) ]))))
  (fresh (a p)
    (prime a p)
    (conde
     [(== p err) (== outcome reject)]
     [(never-equalo p err)
      (Verifier A a p outcome)] ) )]
     [(A p func)
      (qbf_functional_arith_mk
                          A p func)] ))
```

```
;;; Verifier - 3 cases
(define Verifier
 (lambda (A a p outcome)
  (conde
   [(numberl A)
    (conde
     [(== A a) (== outcome accept)]
     [(never-equalo A a) (== outcome reject)] )]
     [(fresh (a1 A2 a2 na)
       (== `(,a1 + ,A2) A)
       (subl a a1 na)
       (modl na p a2)
       (Verifier A2 a2 p outcome) )]
     [(fresh (a1 A2)
       (== `(,a1 * ,A2) A)
       (== a1 (numl 0))
       (conde
        [(== (numl 0) a) (== outcome accept)]
        [(never-equalo (numl 0) a)
         (== outcome reject)]) )]
        [(fresh (a1 A2 a2 na)
          (== `(,a1 * ,A2) A)
          (never-equalo a1 (numl 0))
          (divl a a1 na)
          (modl na p a2)
          (Verifier A2 a2 p outcome) )]
        [(fresh (q q0 q1 ta a-check r nA na qr)
          (Prover A p q)
          (qbf_apply_func_mk q (numl 0) q0)
          (qbf_apply_func_mk q (numl 1) q1)
          (conde
           [(caro A 'sum) (addl q0 q1 ta)]
           [(caro A 'product) (mull q0 q1 ta)] )
          (modl ta p a-check)
           (conde
            [(== a a-check)
             (== r (numl (get-rand-prime-up)))
             (qbf_randomize_arith_mk A r p nA)
             (qbf_apply_func_mk q r qr)
             (modl qr p na)
             (Verifier nA na p outcome) ]
            [(never-equalo a a-check)
             (== outcome reject)] ))] )))
```

Table 2: Our *miniKanren* implementation of IPS

5 Experiments and Analysis

Experiments are carried out on a 1.79GHz AMD Turion(tm) 64 mobile processor. As should be expected, test cases return the expected (right) results. However, mK tests appear to be either taking a lot longer or diverging. This is due to the non-deterministic behavior of relational conditional clauses such as *conde*. There really is no way to get rid of this behavior unless we introduce some special constraints that will reduce the totality of this scheme's reversibility; which is not what we want to accomplish. Based on some modifications of the testing "environment", we notice several behaviors which are not too far from expected.

Smaller number of primes in look-up table results in faster convergence. First, by reducing the number of prime numbers in the look-up table, we are able to get tests cases to converge faster, with the right results, and especially if they are supposed to result in a rejecting state. This is due to the reduction in depth (and length) of the non-deterministic tree associated with the behavior of the function runs.

Using *fake-/o* as opposed to */o* results in even faster convergence. A fake implementation of */o*, which uses *project* and unfortunately removes the reversible behavior observed with the real */o*, also definitely speeds up the convergence; given that the *Scheme*-related tests are converging with no noticeable problem. The following test case runs much more efficiently when using fake-/o, rather than /o, under either large prime numbers ("large-primes.ss") or smaller prime numbers ("primes.ss").

```
(define A6t_qbf '(exists x1
                  (exists x2
                    (((x1 or (not x1)) or
                      (x2 or (not x2))) or
                     (x2 or (not x2))))) )

(define A6t_qbf (label-vars (qbf_simplify A6t_qbf)))

("\nA6t_qbf \n" (time
                   (run* (q)
                     (IPsys_mk A6t_qbf q))))
```

Running it with large prime numbers, here are the outputs generated with /o and fake-/o:

```
using /o
------------------------
A6t_qbf
 (time (run* (q) ...))
    130 collections
    28844 ms elapsed cpu time, including 563 ms collecting
```

```
    29703 ms elapsed real time, including 595 ms collecting
    141343520 bytes allocated, including 142543400 bytes reclaimed
(1)

using fake-/o
------------------------
A6t_qbf
 (time (run* (q) ...))
    20 collections
    1187 ms elapsed cpu time, including 0 ms collecting
    1219 ms elapsed real time, including 0 ms collecting
    21350328 bytes allocated, including 23036440 bytes reclaimed
(1)
```

Alternatively, here are the generated outputs for smaller prime numbers.

```
using /o
------------------------
A6t_qbf
 (time (run* (q) ...))
    39 collections
    8078 ms elapsed cpu time, including 296 ms collecting
    8094 ms elapsed real time, including 297 ms collecting
    42758784 bytes allocated, including 45033504 bytes reclaimed
(1)

using fake-/o
------------------------
A6t_qbf
 (time (run* (q) ...))
    14 collections
    1641 ms elapsed cpu time, including 265 ms collecting
    1640 ms elapsed real time, including 266 ms collecting
    15423264 bytes allocated, including 17572624 bytes reclaimed
(1)
```

Clearly, *fake-/o* produces a better runtime efficiency than */o*. However, relying of it defeats the purpose of our experiment which requires total reversibility to account for all possible variants of IPS.

Smaller prime numbers are more favorable for our "signed" mK representation. Another major part of mK's non-deterministic behavior is spent computing modulo arithmetic; each operation traversing long lists representing large numbers. Again, the depth (and length) of this behavior can be reduced simply by using smaller primes. The expected change in efficiency is again observed with the test case, using */o*, presented above.

Reversible cases: format the output to get most correct inputs. Another remark with these reversible test cases is the fact that, for a given outcome of "accept" or "reject", there are multiple possible results. Nevertheless, we notice that due to very early unifications, most of the results are not as helpful as we would wish for. However, we can restrict the resulting QBF to satisfy specific formats. Below are some running test cases and their results.

```
----------------------------------------------------------
  ("\naccept \n" (run 1 (q)
                   (fresh (x y)
                     (== `(,x (var x)  ,y) q)
                     (IPsys_mk q accept))))
accept
 ((exists (var x) (var x)))
----------------------------------------------------------
  ("\nreject \n" (run 1 (q)
                   (fresh (x y)
                     (== `(,x (var x)  ,y) q)
                     (IPsys_mk q reject))))

reject
 ((forall (var x) (exists (var x) (var x))))
----------------------------------------------------------
  ("\naccept \n" (run 1 (q)
                   (fresh (x y z)
                     (== `(,x (var x) (,y (var y) ,z)) q)
                     (IPsys_mk q accept))))
accept
 ((exists (var x) (exists (var y) (var y))))
----------------------------------------------------------
  ("\nreject \n" (run 1 (q)
                   (fresh (x y z)
                     (== `(,x (var x) (,y (var y) ,z)) q)
                     (IPsys_mk q reject))))
reject
 ((forall (var x) (forall (var y) (var x))))
----------------------------------------------------------
  ("\naccept \n" (run 1 (q)
                   (fresh (x y)
                     (== `(,x (var x)  (var ,y)) q)
                     (IPsys_mk q accept))))
accept
 ((exists (var x) (var x)))
----------------------------------------------------------
  ("\nreject \n" (run 1 (q)
                   (fresh (x y)
                     (== `(,x (var x)  (var ,y)) q)
                     (IPsys_mk q reject))))
```

```
reject
 ((forall (var x) (var x)))
------------------------------------------------------------
```

6 Future work

Throughout this experiment, we have noticed limitations in the prime numbers' generation as well as in the efficiency of arithmetic operations with those prime numbers themselves. A step further in this experimentation could then involve finding (and implementing) a very efficient algorithm for generating primes numbers indefinitely (and defining a good divergence detection constant), and finding an acceptable compromise between the number of prime numbers needed, to solve the problem of deciding the truth of QBF using our IPS, and the time spend performing computations on those primes. Another step further would be to come up with a *miniKanren* version of *unique-name* as defined in "Utils/match.ss" or "*symbolize**" as defined in "qbf.ss".

7 Conclusion

We have presented a *miniKanren* implementation of an interactive proof system for deciding the truth of (simple) quantified boolean formula. We went from acquiring some understanding of QBF, to discovering an IPS for deciding the truth of QBF as defined by Adi Shamir in 1992, to providing both one-way(*Scheme*) and two-way (mK) implementations of that IPS. we have successfully observed an enjoyable runtime efficiency with the *Scheme* implementation. However, we still need to think about some strategies that would boost up our mK implementation.

Nevertheless, we already have some elegant library tools set up that can be useful even outside the scope of this experiment. We have also accomplished our goal to provide ourselves a set up for all variants of IPS. In the main time, *miniKanren* implementation strategization is in order.

Acknowledgment

I am greatly indebted to all those who directly or indirectly contributed to this work. I particularly thank William Byrd for all the helpful input and attention during the implementation process. I would also like to thank Kent Dybvig, Oscar Waddell, Daniel P. Friedman, Erik Hilsdale, Steve Ganz and William Byrd, for making a good portion of the helper tools used in this implementation available ("match.ss", "helpers.ss", "mk.ss", etc...). I am also indebted to Amr Sabry for his helpful pointers in the process of understanding the proof of IP = PSPACE.

References

[1] Shamir, A. 1992. *IP = PSPACE*. J. ACM 39, 4 (Oct. 1992), 869-877. DOI= http://doi.acm.org/10.1145/146585.146609

[2] Hartmanis, J., R. Chang, D. Ranjan, P. Rohatgi. 1990. *On IP = PSPACE and Theorems with Narrow Proofs*.

[3] Goldwasser, S., S. Micali, C. Rackoff. 1985. *The Knowledge complexity of interactive proof-systems*. Proceedings of 17th ACM Symposium on the Theory of Computation, Providence, Rhode Island. pp. 291-304.

[4] Voufo. 2006. "Quantum Complexities and Interactive Proof Systems". [http://www.cs.indiana.edu/~lvoufo/project.pdf]

[5] Voufo. 2006. "Quantum Complexity Classes". [http://www.cs.indiana.edu/~lvoufo/qcc.pdf]

[6] Dybvig, K. *The Scheme Programming Language*. Third Ed. [http://www.scheme.com/tspl3/]

[7] Dybvig, K. *Chez Scheme Version 7 User's Guide*. [http://www.scheme.com/csug7/]

[8] Abelson, H. et al. 1998. *Revised^5 Report on the Algorithmic Language Scheme*. [http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs.html]

[9] Friedman, D., W. Bird, O. Kiselyov. 2005. *The Reasoned Schemer*. [http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10663]

A Overview on Interactive Proof Systems

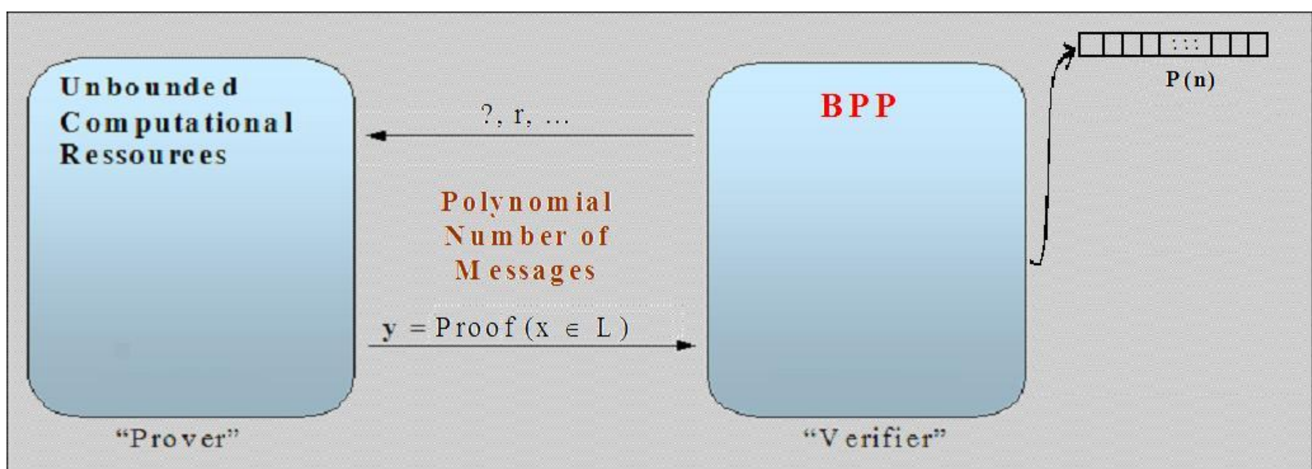

Figure 1: A graphical description of an Interactive Proof System

A.1 Brief History

Interactive Proof Systems (IPS) were first introduced in 1985 by Goldwasser et al.[3] They consist of an all-powerful (unbounded computational resources) Turing Machine (TM) called a prover (P), and of a probabilistic polynomial-time machine called a Verifier (V). V has access to a random bit string not visible to P, and whose length is polynomial on the input size. In addition, both P and V originally receive the same input string. Then they engage into a series of question-answers where P continuously presents a proof that an input w is in some language L, and V checks that the presented proof is in fact correct. The interaction completes after a polynomial number of messages (relative to the input size), and V must decide whether or not w is in the language with only a 1/3 chance of error. Of course, V must also ensure that it is not fooled by a dishonest (lying) Prover(Q) in the process.

Laszlo Babai, later on, introduced a variation of this scheme called the Merlin-Arthur protocol, in which Merlin is the prover and Arthur is the verifier. This differs from the previous in that the number of rounds of interaction is bounded by a constant, rather than a polynomial.

A.2 Definition:

The above introduces the complexity class called IP. More formally, a problem is in the class IP if it is solvable by an Interactive Proof System. By definition:

For any language L, $L \in IP \Leftrightarrow \exists V, P | \forall Q, w$:

- ▷ $w \in L \Leftrightarrow Pr[V \leftrightarrow P \, accepts \, w] \geq 2/3$[8]
- ▷ $w \notin L \Leftrightarrow Pr[V \leftrightarrow Q \, accepts \, w] \leq 1/3$

A.3 Variants

There exists multiple variations of interactive proof systems. The following are just a few. [9].

A.3.1 MIP:

the class MIP is described by powerful interactive proof system based on IP, in which there are two independent provers, who cannot communicate with each other once the verifier begins sending messages. This makes it easier to detect a malicious prover trying to trick the verifier, since there is another prover that it can double-check with. So far, we know that MIP = NEXPTIME. Also, all languages in NP have zero-knowledge proofs in an MIP system,

---

[8]It is important to notice that, more formally, the delimiting constant (currently "2/3") should be "$1/2+\delta$", where $\delta > 0$. It just happens that "2/3" is the most commonly used one.

[9]The proof of $IP = PSPACE$ not only placed the IP class in the standard classification of feasible computations, but also provided alternative definitions of classical (and quantum) complexity classes.

Refer to the following for a brief overview:

* Voufo. 2006. "Quantum Complexities and Interactive Proof Systems". [http://www.cs.indiana.edu/~lvoufo/project.pdf]

* Voufo. 2006. "Quantum Complexity Classes". [http://www.cs.indiana.edu/~lvoufo/qcc.pdf]

without any additional assumptions; This is only known for IP assuming the existence of one-way functions.

A.3.2 IPP: Unbounded IP

This is a variant of IP where we replace the BPP verifier by a PP verifier. It modifies the completeness and soundness conditions as follows:

- ▷ Completeness:

  If a string is in the language, the honest verifier will be convinced of this fact by an honest prover with probability at least 1/2.

- ▷ Soundness:

  If the string is not in the language, no prover can convince the honest verifier that it is in the language, except with probability less than 1/2.

Currently, we know that IPP = PSPACE. However, let us notice that while IPP=PSPACE with respect to all oracles, IP ≠ PSPACE with respect to almost all oracles.

A.3.3 compIP: competitive IP

A compIP system weakens the completeness condition in a way that weakens the prover, while IPP and QIP give more power to the verifier:

- ▷ Completeness:

  If a string is in the language L, the honest verifier will be convinced of this fact by an honest prover with probability at least 2/3. Moreover, the prover will do so in probabilistic polynomial time given access to an oracle for the language L.

Here, the prover is a BPP machine with access to an oracle for the language, but only in the completeness case, not the soundness case. If a language is in compIP, then interactively proving it is in some sense as easy as deciding it. With the oracle, the prover can easily solve the problem, but its limited power makes it much more difficult to convince the verifier of anything. compIP isn't even known or believed to contain NP.

However, it can solve some problems believed to be hard. For instance, problems such as the graph (non)isomorphism and quadratic non-residuosity are in compIP (oracles).

B The Proof of IP=PSPACE

The proof for $IP = PSPACE$ consists of two main subproofs: $IP \subseteq PSPACE$ and $PSPACE \subseteq IP$.

Proving that $IP \subseteq PSPACE$ can be trivially done by presenting a simulation of an IPS by a polynomial space machine (cf. C).

The proof for $PSPACE \subseteq IP$ is credited to Adi Shamir[1] who presented a reduction of the problem into: finding a PSPACE-complete problem, and then proving that that problem

is in IP. This said, we realize that the problem of deciding the truth of Quantified Boolean Formulae (QBF) has been known to be PSPACE-Complete. Hence, showing that it is in IP will constitute our proof. In other words, let's represent the problem of deciding the truth of QBF with TQBF. Then, $(TQBF \in PSPACE - Complete)$ and $(TQBF \in IP) \Rightarrow PSPACE \subseteq IP$.

We can then show that $TQBF \in IP$ simply by illustrating the existence of an interactive proof system for solving TQBF, as in the part 2.2.

C Simulating an Interactive Proof System by a PSPACE machine

- ▷ Consider A, a language in IP.
- ▷ By the definition of an IP system, we know that:

  On input w with length n, A's verifier V exchanges exactly $p = f(n)$ messages, based on a random number r gotten from its random string, and given a function f that is polynomial in n.
- ▷ This said, let's **construct a** PSPACE **machine** M **that simulates** V.

  To do this, we define M as follows:

  $\Pr[M \text{ accepts } w] = max_p \Pr[V \leftrightarrow P \text{ accepts } w]$

  That is, the probability that M accepts an input w corresponds to the maximum of all the probabilities that the interaction of V and P would accept w given a possible value of r. Remember that there are p possible values for r ($p = size\ of\ r$).
- ▷ By the definition of IP,

  $\Pr[M \text{ accepts } w] \geq 2/3$ if $w \in L$.

  $\Pr[M \text{ accepts } w] \leq 1/3$ if $w \notin L$.
- ▷ Now, let's **verify that the value of** $\Pr[M \text{ accepts } w]$ **can in fact be calculated in** PSPACE.
  - ► Let M_j be the sequence of messages, $m_1\#m_2\#...\#m_j$, exchanged by P and V.
  - ► And let's **generalize the interaction of** V **and** P **to start with an arbitrary message stream** M_j as follows:

    $(V \leftrightarrow P)(w, r, M_j) =$ accept, if M_j can be extended with the messages m_{j+1} through m_p such that:

    1. For $j \leq i < p$, where i is even, $V(w, r, M_j) = m_{i+1}$ ($\Rightarrow V$ sends message)
    2. For $j \leq i < p$, where i is odd, $P(w, r, M_j) = m_{i+1}$ ($\Rightarrow P$ sends message)
    3. The final message m_p in the message history is *accept* (final message is to *accept*)

Notice that items (1) and (2) ensure the validity of M_j and item (3) ensures that M_j leads to an accepting state)

- From this, we gather the following generalization:
  $\Pr[V \leftrightarrow P \text{ accepts } w \text{ starting at } M_j] = \Pr[(V \leftrightarrow P)(w, r, M_j) = \text{accept}]$
- **Generalizing the earlier definition of M further**, we get:
  $\Pr[M \text{ accepts } w \text{ starting at } M_j] = max_p Pr\,[V \leftrightarrow P \text{ accepts } w \text{ starting at } M_j]$

Therefore, our verification step now consists of proving that the value of

$\Pr[M \text{ accepts } w \text{ starting at } M_j]$ can be computed in $PSPACE$.

▷ To do that, let's **define the function N_{M_j}, for every $0 \leq j \leq p$ and every message history M_j**[10], such that:

$$N_{M_j} = \begin{cases} 0 & \text{if } j = p \text{ and } m_p = \textit{reject}. \\ 1 & \text{if } j = p \text{ and } m_p = \textit{accept}. \\ {}_{m_{j+1}} N_{M_{j+1}} & \text{if } j < p \text{ and } j \text{ is odd.} \\ {}_{m_{j+1}} N_{M_{j+1}} & \text{if } j < p \text{ and } j \text{ is even.} \end{cases},$$

where:

- Let $\Pr_r$ = probability taken over a particular random value r.
- Then, $wt{-}avg_{m_{j+1}} N_{M_{j+1}} = \sum_{m_{j+1}} (\Pr_r[V(x, r, M_j) = m_{j+1}]\, N_{M_{j+1}})$ = the average of $N_{M_{j+1}}$, weighted by the probability that V sends message m_{j+1}.

To better understand this, remember that in an IP system,

- On a given input, V sends and receives a polynomial number of messages to and from P, each time with a probability that depends on the randomly chosen r.
- V makes a decision based on the maximum of all the probabilities of accepting messages. Therefore, the only way for P to ensure that its message is accepted is by making its probability equal that maximum probability.
- Alternatively, P makes a decision based on averaging all the probabilities of receiving messages[11]. Hence, in order for V to ensure that its message is accepted, it has to ensure that its probability is equal to that average.

▷ We notice that any attempt to layout the "values" of N_{M_j} suggests that it describes a $PSPACE$ machine. However, to be certain of that, let's consider M_0, and **show that N_{M_0} can be computed in polynomial space**.

This entails simply noticing that, to compute N_{M_0}, an algorithm can recursively calculate the values N_{M_j} for every j in M_j. Moreover, since the depth of the recursion is p, only polynomial space is necessary.

---

[10] Notice that a probability at M_0 means a probability before starting to "read" the p messages.

[11] Notice that this operation (average) could be anything since the P is set up to accept any message from V. "Averaging" simply works as means of "dont-care" on inputs from V.

- ▷ Now that we know that N_{M_j} does in fact describe a $PSPACE$ machine, let's **verify that the value of** $\Pr[M \text{ accepts } w \text{ starting at } M_j]$**, and thus that of** $\Pr[M \text{ accepts } w]$**, can be computed in** $PSPACE$**, through** N_{M_j}.
- ▷ This entails showing that: $N_{M_0} = \Pr[M \text{ accepts } w]$.
- ▷ For that, we must show the following:

  For every $0 \leq j \leq p$, and every M_j, $N_{M_j} = Pr[M \text{ accepts } w \text{ starting at } M_j]$
- ▷ **Proof by induction on** j:
  - ▶ **Base case**: $j = p$
    - ∗ m_p is either *accept* or *reject.*
    - ∗ If m_p is *accept,*
      then $N_{M_j} = 1$ by definition of N_{M_j},
      and $\Pr[M \text{ accepts } w \text{ starting at } M_j] = 1$ since the message stream indicates acceptance.
    - ∗ A similar argument holds for the case where m_p is *reject.*
  - ▶ **Induction Hypothesis (on** $k \geq 0$**)**:
    $k = j + 1 \leq p$,
    and $N_{M_{j+1}} = \Pr[M \text{ accepts } w \text{ starting at } M_{j+1}]$.
  - ▶ **Prove true at** $k - 1$:
    $N_{M_j} = \Pr[M \text{ accepts } w \text{ starting at } M_j]$
    - ∗ If j is even, then:
      - m_{j+1} is a message from P to V.
      - By def. of N_{M_j}, $N_{M_j} = \sum_{m_{j+1}} (\Pr_r [V(x, r, M_j) = m_{j+1}]\, N_{M_{j+1}})$.
      - By IH, $N_{M_j} = \sum_{m_{j+1}} (Pr_r [V(x, r, M_j) = m_{j+1}] \cdot \Pr[M \text{ accepts } w \text{ starting at } M_{j+1}])$.
      - By def., $N_{M_j} = \Pr[M \text{ accepts } w \text{ starting at } M_j]$.
    - ∗ If j is odd, then:
      - m_{j+1} is a message from V to P.
      - By def. of N_{M_j}, $N_{M_j} = {}_{m_{j+1}} N_{M_{j+1}}$.
      - By IH, $N_{M_j} = {}_{m_{j+1}} \cdot \Pr[M \text{ accepts } w \text{ starting at } M_{j+1}]$.
      - $N_{M_j} = \Pr[M \text{ accepts } w \text{ starting at } M_j]$, since:
        1. ${}_{m_{j+1}} \cdot \Pr[M \text{ accepts } w \text{ starting at } M_{j+1}] \leq \Pr[M \text{ accepts } w \text{ starting at } M_j]$, since the prover on the right-hand side could send the message m_{j+1} to maximize the expression on the left-hand side.
        2. ${}_{m_{j+1}} \cdot \Pr[M \text{ accepts } w \text{ starting at } M_{j+1}] \geq \Pr[M \text{ accepts } w \text{ starting at } M_j]$, since the same prover cannot do any better than send that same message.

From the above, we get:

$$_{m_{j+1}} \cdot \Pr\left[M \text{ accepts } w \text{ starting at } M_{j+1}\right] = \Pr\left[M \text{ accepts } w \text{ starting at } M_j\right]$$

So, $N_{M_j} = Pr[M$ accepts w starting at $M_j]$ and N_{M_j} is a $PSPACE$ machine.

▷ Therefore, $\Pr[M$ accepts w starting at $M_j]$ is a $PSPACE$ machine.

▷ Thus, $\Pr[M$ accepts $w]$ is a $PSPACE$ machine.

▷ So, the machine M does simulate an V in an IPS.

Hence, $IP \subseteq$ PSPACE.

D The *mk_ arithmetic* module

```
;;;-------------------------------------
;;;  signed arithmetic with mK's bignums
;;;    (will be extended to other ops as needed)
;;;  --> labels numbers with their signs (pos or neg)
;;;-------------------------------------
(module mk_arithmetic
  ( pos-label neg-label numl numberl
    zerol posl negl negatel
    addl subl mull divl
    quotl reml modl    ;; logl expl
    fake-divl fake-quotl fake-reml fake-modl)

  ;;; label for positive numbers
  (define pos-label 'pos)

  ;;; label for negative numbers
  (define neg-label 'neg)

  ;;; constructor for a number
  ;;; --> 0 is originally positive
  (define-syntax numl
    (syntax-rules ()
      [(_ numb)
       (let [(number numb)]
         (if (negative? number)
             `(,neg-label ,(build-num (- number)))
             `(,pos-label ,(build-num number)) ))] ))

  ;;; tests whether given number is zero
  (define zerol
```

```
  (lambda (number)
    (conde
      [(== `(,pos-label ,(build-num 0)) number)]
      [(== `(,neg-label ,(build-num 0)) number)])))

;;; tests whether given number is positive
(define-syntax posl
  (syntax-rules ()
    [(_ number) (caro number pos-label)]))

;;; tests whether given number is negative
(define-syntax negl
  (syntax-rules ()
    [(_ number) (caro number neg-label)]))

;;; Is this a number?
(define numberl
  (lambda (number)
    (conde
      [(posl number)]
      [(negl number)])))

;;; negate given number
(define negatel
  (lambda (number neg-number)
    (fresh (numb)
      (conde
        [(== `(,pos-label ,numb) number)
         (== `(,neg-label ,numb) neg-number)]
        [(== `(,neg-label ,numb) number)
         (== `(,pos-label ,numb) neg-number)]))))

;;; add two numbers
(define addl
  (lambda (number1 number2 sum)
    (fresh (sign1 numb1 sign2 numb2 numb)
      (== `(,sign1 ,numb1) number1)
      (== `(,sign2 ,numb2) number2)
      (conde
        [(== sign2 sign1)
         (pluso numb1 numb2 numb)
         (== `(,sign1 ,numb) sum)]
        [(never-equalo sign2 sign1)
         (conde
           [(<o numb1 numb2)
            (minuso numb2 numb1 numb)
            (== `(,sign2 ,numb) sum)]
```

```
          [(<o numb2 numb1)
           (minuso numb1 numb2 numb)
           (== `(,sign1 ,numb) sum)]
          [(eqo numb1 numb2)
           (== `(,pos-label ,(build-num 0)) sum)] )] )
      )))

;;; subtract two numbers
(define subl
  (lambda (number1 number2 diff)
    (fresh (neg-number2)
      (negatel number2 neg-number2)
      (addl number1 neg-number2 diff))))

;;; nultiply two numbers
(define mull
  (lambda (number1 number2 prod)
    (fresh (sign1 numb1 sign2 numb2 numb)
      (== `(,sign1 ,numb1) number1)
      (== `(,sign2 ,numb2) number2)
      (*o numb1 numb2 numb)
      (conde
        [(== sign2 sign1)
         (== `(,pos-label ,numb) prod)]
        [(never-equalo sign2 sign1)
         (== `(,neg-label ,numb) prod)] ))))

;;; divide two numbers
(define divl
  (lambda (num den quotient remainder)
    (fresh (sign1 numb1 sign2 numb2 quot rem)
      (== `(,sign1 ,numb1) num)
      (== `(,sign2 ,numb2) den)
      (/o numb1 numb2 quot rem)
      (== `(,sign1 ,rem) remainder)
      (conde
        [(== sign2 sign1) (== `(,pos-label ,quot) quotient)]
        [(never-equalo sign2 sign1) (== `(,neg-label ,quot) quotient)] ))))

;;; return the quotient of the division of two numbers
(define quotl
  (lambda (num den quotient)
    (fresh (sign1 numb1 sign2 numb2 quot rem)
      (== `(,sign1 ,numb1) num)
      (== `(,sign2 ,numb2) den)
      (/o numb1 numb2 quot rem)
      (conde
```

```
        [(== sign2 sign1) (== `(,pos-label ,quot) quotient)]
        [(never-equalo sign2 sign1) (== `(,neg-label ,quot) quotient)] ))))

;;; return the remainder of the division of two numbers
(define reml
  (lambda (num den remainder)
    (fresh (sign1 numb1 sign2 numb2 quot rem)
      (== `(,sign1 ,numb1) num)
      (== `(,sign2 ,numb2) den)
      (/o numb1 numb2 quot rem)
      (== `(,sign1 ,rem) remainder) )))

;;; return a positive value for "number `mod` modulo"
(define modl
  (lambda (num den mod)
    (fresh (r)
      (reml num den r)
      (conde
        [(negl r) (posl den) (addl r den mod)]
        [(negl r) (negl den) (subl den r mod)]
        [(posl r) (== r mod)]) )))

(define olegnum->schemenum
  (lambda (on)
    (cond
      [(null? on) 0]
      [else (string->number
              (format "#b~a" (apply string-append
                                 (map number->string (reverse on)))))])))

(define fake-/o
  (lambda (numb1 numb2 quot rem)
    (project (numb1 numb2 quot rem)
      (let ((n1 (olegnum->schemenum numb1))
            (n2 (olegnum->schemenum numb2)))
        (let ((q (quotient n1 n2))
              (r (modulo n1 n2)))
          (fresh ()
            (== (build-num q) quot)
            (== (build-num r) rem)))))))

(define fake-divl
  (lambda (num den quotient remainder)
    (fresh (sign1 numb1 sign2 numb2 quot rem)
      (== `(,sign1 ,numb1) num)
      (== `(,sign2 ,numb2) den)
      (fake-/o numb1 numb2 quot rem)
```

```
        (== `(,sign1 ,rem) remainder)
        (conde
          [(== sign2 sign1) (== `(,pos-label ,quot) quotient)]
          [(never-equalo sign2 sign1) (== `(,neg-label ,quot) quotient)] ))))

;;; return the quotient of the division of two numbers
(define fake-quotl
  (lambda (num den quotient)
    (fresh (sign1 numb1 sign2 numb2 quot rem)
      (== `(,sign1 ,numb1) num)
      (== `(,sign2 ,numb2) den)
      (fake-/o numb1 numb2 quot rem)
      (conde
        [(== sign2 sign1) (== `(,pos-label ,quot) quotient)]
        [(never-equalo sign2 sign1) (== `(,neg-label ,quot) quotient)] ))))

;;; return the remainder of the division of two numbers
(define fake-reml
  (lambda (num den remainder)
    (fresh (sign1 numb1 sign2 numb2 quot rem)
      (== `(,sign1 ,numb1) num)
      (== `(,sign2 ,numb2) den)
      (fake-/o numb1 numb2 quot rem)
      (== `(,sign1 ,rem) remainder) )))

;;; return a positive value for "number `mod` modulo"
(define fake-modl
  (lambda (num den mod)
    (fresh (r)
      (fake-reml num den r)
      (conde
        [(negl r) (posl den) (addl r den mod)]
        [(negl r) (negl den) (subl den r mod)]
        [(posl r) (== r mod)]) )))
)
```

E Where is the "*schemish*" folder?

http://www.cs.indiana.edu/~lvoufo/schemish.