Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®
The Journal of Logic and
Algebraic Programming 56 (2003) 3–21

THE JOURNAL OF
LOGIC AND
ALGEBRAIC
PROGRAMMING

www.elsevier.com/locate/jlap

# Verification of the Miller–Rabin probabilistic primality test

Joe Hurd [1]

*Computer Laboratory, University of Cambridge, Cambridge, UK*

**Abstract**

Using the HOL theorem prover, we apply our formalization of probability theory to specify and verify the Miller–Rabin probabilistic primality test. The version of the test commonly found in algorithm textbooks implicitly accepts probabilistic termination, but our own verified implementation satisfies the stronger property of guaranteed termination. Completing the proof of correctness requires a significant body of group theory and computational number theory to be formalized in the theorem prover. Once verified, the primality test can either be executed in the logic (using rewriting) and used to prove the compositeness of numbers, or manually extracted to standard ML and used to find highly probable primes.


*Keywords:* Formal verification; Random algorithms; Primality test

## 1. Introduction

In the 1970s a handful of probabilistic algorithms were introduced that demonstrated two practical advantages over deterministic alternatives: simplicity of expression and efficiency of execution. An algorithm of Berlekamp [3] uses randomization to factor polynomials; Solovay and Strassen [18] introduced a probabilistic primality test based on the Jacobi symbol; and Rabin [16] presented two probabilistic algorithms: the first finds the nearest neighbours of a set $S \subset \mathbb{R}^n$, and the second uses a number theory result of Miller [14] to test numbers for primality.

This last algorithm has subsequently become known as the Miller–Rabin probabilistic primality test, and is a fast way to test large numbers for primality. In his 1976 paper, Rabin evaluates the algorithm by finding the largest (probable) prime less than $2^{400}$ (it takes less than a minute to return the result $2^{400} - 593$), and reports that "the algorithm was also used to find twin primes by far larger than any hitherto known pair." Today the probabilistic primality test is used in computer algebra systems such as Mathematica, and it is also relevant to public key cryptography software (the RSA algorithm requires a modulus

[1] Supported by EPSRC project GR/R27105/01.

1567-8326/02/$ - see front matter 
doi:10.1016/S1567-8326(02)00065-6

of the form $n = pq$ where $p$ and $q$ are primes). Surprisingly, the popular email encryption program PGP (and the Gnu version GPG) use the Fermat test to check numbers for primality, although the Miller–Rabin test is stronger and involves no extra computation.

In this paper we report on using the HOL theorem prover to formally prove correctness of the Miller–Rabin probabilistic primality test. Concretely, we define in HOL a function miller_rabin that implements the probabilistic test, and formally prove the following two properties.

**Theorem 1.** *Correctness of the Miller–Rabin primality test*

$$\vdash \forall n, t, s.\ \mathsf{prime}\ n \ \Rightarrow\ \mathsf{fst}\ (\mathsf{miller\_rabin}\ n\ t\ s) = \top \tag{1}$$

$$\vdash \forall n, t.\ \neg\mathsf{prime}\ n \ \Rightarrow\ 1 - 2^{-t} \leqslant \mathbb{P}\{s : \mathsf{fst}\ (\mathsf{miller\_rabin}\ n\ t\ s) = \bot\} \tag{2}$$

The fst in the statements simply picks out the result of the test (see Section 3.1 for a description of our model of probabilistic programs). The miller_rabin test takes two natural number parameters $n$ and $t$ (in addition to a source $s$ of random bits), where $n$ is the number that we wish to test for primality and $t$ determines the amount of computation that the test is allowed to perform. If $n$ is prime then the test is guaranteed to return $\top$; if $n$ is composite then it will return $\bot$ with probability at least $1 - 2^{-t}$. Thus for a given value of $n$ if miller_rabin $n\ t\ s$ returns $\bot$ then $n$ is definitely composite, but if it returns $\top$ then all we know is that $n$ is probably prime. [2] However, setting $t = 50$ we see that the probability of the algorithm returning $\top$ for an $n$ that is actually composite is $\leqslant 2^{-50} < 10^{-15}$.

The verification of the Miller–Rabin primality test was checked using the HOL theorem-prover. This is the first published verification in a theorem prover of a commercially used algorithm with a probabilistic specification. We build on earlier work [11] and show that the formal probability framework we laid out there is up to the challenge. In addition, the verified implementation of the primality test that we describe here improves on the more abstract version commonly found in algorithm textbooks. There, a generator is generally assumed that can produce uniformly distributed random numbers in the range $\{0, \ldots, n-1\}$, while we assume only a generator of random bits. [3] The difference is that while high quality random bits are provided by most operating systems (e.g., in Linux from `/dev/random`), the generation of uniformly distributed random numbers from these random bits is slightly delicate and requires termination with probability 1. In contrast, our implementation is guaranteed to terminate on all inputs.

Completing the proof of correctness requires a significant body of group theory and computational number theory to be formalized in the theorem prover, and Section 2 shows how the classical results fit together in the verification. This formalization actually constituted the bulk of the effort and provided the testing ground for a new automatic proof procedure; we briefly report on this experience. Section 3 describes the somewhat easier task of interfacing the number theory with the probability theory to produce the result, and we also give an immediate application in the form of a procedure for formally proving that numbers are composite. Finally, we highlight the software engineering benefit of this formal methods research by manually extracting our Miller–Rabin primality test to the ML programming language. In

[2] Making precise that "probably" is a hard problem: the probability that $n$ is prime given that miller_rabin $n\ t\ s$ returned $\top$ depends on the set $S$ from which $n$ was chosen and the distribution of primes in $S$.

[3] In this paper, a *"generator of random bits"* means an infinite sequence of IID Bernoulli(1/2) random variables.

Section 4 we examine the correctness issues in extracting the algorithm to ML, and profile its performance. Finally in Sections 5 and 6 we conclude and look at related work.

### *1.1. Notation*

We use sans serif font to notate higher-order logic constants, such as the function fst that picks the first component of a pair, the natural number predicate prime, the greatest common divisor function gcd, and the group predicate cyclic. For standard mathematical functions we use mathematical font: examples are addition ($a + b$), the remainder function ($a$ mod $b$), function composition ($g \circ f$) and Euler's totient function ($\phi(n)$). We rely on context to disambiguate $|S|$ to mean the cardinality of the set $S$, $|g|$ to mean the order of the group element $g$, and $a \mid b \mid c$ to mean that both $a$ divides $b$ and $b$ divides $c$. Note also that $a \not| b$ means that $a$ does not divide $b$. When doing informal mathematics, we follow the convenient custom of confusing the group $G$ with its carrier set; in HOL theorems we explicitly write set $G$ for the carrier set (and $*_G$ for the operation).

## 2. Computational Number Theory

### *2.1. Definitions*

Our HOL implementation of the Miller–Rabin algorithm is (almost) a functional translation of the version presented in Cormen et al. [6]. To prepare, we define functions to factor out powers of 2 and perform modular exponentiation. We omit the definitions, but here are the correctness theorems for these functions:

$$\vdash \forall n, r, s. \quad 0 < n \Rightarrow (\mathsf{factor\_twos}\ n = (r, s) \iff \mathsf{odd}\ s \wedge 2^r s = n) \tag{3}$$

$$\vdash \forall n, a, b.\ 1 < n \Rightarrow \mathsf{modexp}\ n\ a\ b = (a^b \bmod n) \tag{4}$$

We use these two functions to define a (non-probabilistic) function witness $a\ n$, that returns $\top$ if the base $a$ can be used to provide a quick proof that $n$ is composite, and $\bot$ otherwise. We assume that $a$ and $n$ satisfy $0 < a < n$. The witness function uses a helper function witness_tail to calculate $a^{n-1} \bmod n$, performing some tests for primality along the way; a more detailed explanation follows the definitions.

**Definition 2.** The Miller–Rabin witness function

$$\begin{aligned}
&\vdash \forall n, a, r. \\
&\quad \mathsf{witness_tail}\ n\ a\ 0 = a \neq 1 \wedge \\
&\quad \mathsf{witness_tail}\ n\ a\ (\mathsf{suc}\ r) = \\
&\qquad \mathsf{let}\ a' \leftarrow a^2 \bmod n \\
&\qquad \mathsf{in\ if}\ a' = 1\ \mathsf{then}\ a \neq 1 \wedge a \neq n - 1\ \mathsf{else\ witness_tail}\ n\ a'\ r
\end{aligned} \tag{5}$$

$$\begin{aligned}
&\vdash \forall n, a. \\
&\quad \mathsf{witness}\ n\ a = \\
&\quad \mathsf{let}\ (r, s) \leftarrow \mathsf{factor_twos}\ (n - 1)\ \mathsf{in\ witness_tail}\ n\ (\mathsf{modexp}\ n\ a\ s)\ r
\end{aligned} \tag{6}$$

The witness function calls factor_twos to find $r, s$ such that $s$ is odd and $2^r s = n - 1$, then uses modexp and witness_tail to calculate the sequence

$$\left(a^{2^0 s} \bmod n, a^{2^1 s} \bmod n, \ldots, a^{2^r s} \bmod n\right)$$

This sequence provides two tests for the primality of $n$:

(1) $a^{2^r s} \bmod n = 1$.

(2) If $a^{2^j s} \bmod n = 1$ for some $0 < j \leqslant r$, then either $a^{2^{j-1} s} \bmod n = 1$ or $a^{2^{j-1} s} \bmod n = n - 1$.

Verifying the correctness of the witness function requires us to prove that if $n$ is a prime then both these tests will always be passed. Test 1 is equivalent to $a^{\phi(n)} \bmod n = 1$ (since $2^r s = n - 1 = \phi(n)$ for $n$ prime), and this is exactly Fermat's little theorem. For this reason this test for primality is called the Fermat test. Test 2 is true since for every $x$, if $0 = (x^2 - 1) \bmod n = (x + 1)(x - 1) \bmod n$, then if $n$ is prime we must have that either $(x + 1) \bmod n = 0$ or $(x - 1) \bmod n = 0$. We thus obtain the following correctness theorem for witness:

$$\vdash \forall n, a.\ 0 < a < n \ \wedge\ \mathsf{witness}\ n\ a \ \Rightarrow\ \neg\mathsf{prime}\ n \tag{7}$$

### 2.2. Underlying mathematics

A composite number $n$ that passes a primality test for some base $a$ is called a pseudoprime. In the case of the Fermat test, there exist numbers $n$ that are pseudoprimes for all bases $a$ coprime to $n$. These numbers are called Carmichael numbers, and the two smallest examples are 561 and 1729.[4] Testing Carmichael numbers for primality using the Fermat test is just as hard as factorizing them, since the only bases that fail the test are multiples of divisors.

A theorem of Miller [14] implies that by also performing Test 2, the number of bases that are witnesses for any composite $n$ will be at least $(n - 1)/2$.[5] Therefore combining both tests completely eliminates Carmichael numbers, and Rabin made use of this to implement a probabilistic primality test which tests many bases chosen at random [16]. This algorithm was published in 1976, and is now known as the Miller–Rabin primality test.

In Section 3 we will show how to formally model the probabilistic element of the test. For the rest of the present section, we will describe the formalization of Miller's result that underlies the correctness of the primality test. We first present the informal mathematical proof that the formalization followed, stating along the way the classical results of number theory that are necessary for the result. Then, in Section 2.3, we will examine the interesting problems that arose in the formalization of this number theory result.

[4] 1729 is also famous as the Hardy–Ramanujan number, explained by Snow in the foreword to Hardy's *A Mathematician's Apology* [9]: "Once, in the taxi from London, Hardy noticed its number, 1729. He must have thought about it a little because he entered the room where Ramanujan lay in bed and, with scarcely a hello, blurted out his disappointment with it. It was, he declared, 'rather a dull number,' adding that he hoped that was not a bad omen. 'No, Hardy,' said Ramanujan, 'it is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways'." ($10^3 + 9^3 = 1729 = 12^3 + 1^3$).

[5] In fact, Miller proved the stronger result that the number of nonwitnesses must be at most $\phi(n)/4$, and furthermore this bound can be attained (an example is the Carmichael number 8911).

So first, the formal statement, together with an informal proof:

**Theorem 3.** *Cardinality of Miller–Rabin witnesses*

$$\vdash \forall n. \tag{8}$$
$$1 < n \ \wedge\ \mathsf{odd}\ n \ \wedge\ \neg(\mathsf{prime}\ n) \ \Rightarrow$$
$$n - 1 \leqslant 2\left|\{a : 0 < a < n \ \wedge\ \mathsf{witness\ n\ a}\}\right|$$

**Proof.** We begin with some necessary terminology and some useful properties. Euler's totient function $\phi(n)$ returns the size of the set

$$\{m < n : m \text{ is coprime to } n\}$$

For a prime power $p^a$, it is the case that $\phi(p^a) = p^{a-1}(p-1)$. The multiplicative group $\mathbb{Z}_n^*$ has as elements the numbers $0 < i < n$ coprime to $n$, and the operation is multiplication mod $n$. The Chinese remainder theorem states that for every coprime $p, q$ there is an isomorphism between $\mathbb{Z}_p q^*$ and $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$ (i.e., for every pair of equations ($x$ mod $p$) $= a$ and ($x$ mod $q$) $= b$ there is a solution for $x$, and that solution will be unique mod $pq$).

The proof aims to find a proper subgroup $B$ of the multiplicative group $\mathbb{Z}_n^*$ which contains all the nonwitnesses. This will then imply the result, since by Lagrange's theorem the size of a subgroup must divide the size of the group, and so $|B| \leqslant |\mathbb{Z}_n^*|/2 = \phi(n)/2 \leqslant (n-1)/2$.

Firstly, assume that there exists an $x \in \mathbb{Z}_n^*$ such that $x^{n-1} \bmod n \neq 1$. In this case we choose $B = \{x \in \mathbb{Z}_n^* : x^{n-1} \bmod n = 1\}$. The Fermat test ensures that all nonwitnesses are members of $B$, and since $B$ is closed under multiplication it is a proper subgroup of $\mathbb{Z}_n^*$. Therefore in this case the proof is finished.

Secondly, assume that for every $x \in \mathbb{Z}_n^*$ we have that $x^{n-1} \bmod n = 1$. We next show by contradiction that $n$ cannot be a prime power. If $n = p^a$ (with $p$ prime), then a textbook number theory result shows that the group $\mathbb{Z}_n^*$ is cyclic. This means that there exists an element $g \in \mathbb{Z}_n^*$ with order $|\mathbb{Z}_n^*| = \phi(n) = \phi(p^a) = p^{a-1}(p-1)$. But $g^{n-1} \bmod n = 1$, and so $p^{a-1}(p-1) \mid p^a - 1$. However, this is the required contradiction, since $p \mid p^{a-1}(p-1)$ but $p \not\mid p^a - 1$.

Since $n$ is composite but not a prime power, we can therefore find two numbers $1 < a, b$ with $\gcd(a, b) = 1$ and $ab = n$. Find $r, s$ satisfying $2^r s = n - 1$, with $s$ odd. Next we find a maximal $j \in \{0, \ldots, r\}$ such that there exists a $v \in \mathbb{Z}_n^*$ with $v^{2^j s} \bmod n = n - 1$. Such a $j$ must exist, because since $s$ is odd we can set $j = 0$ and $v = n - 1$. Now choose

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j s} \bmod n = 1 \ \vee\ x^{2^j s} \bmod n = n - 1\}$$

$B$ is closed under multiplication and so is a subgroup of $\mathbb{Z}_n^*$; also the maximality of $j$ ensures that $B$ must contain all nonwitnesses. It remains only to show that $B \neq \mathbb{Z}_n^*$. By the Chinese remainder theorem there exists $w \in \mathbb{Z}_n^*$ such that

$$w \bmod a = v \bmod a \ \wedge\ w \bmod b = 1$$

and so

$$w^{2^j s} \bmod a = a - 1 \ \wedge\ w^{2^j s} \bmod b = 1$$

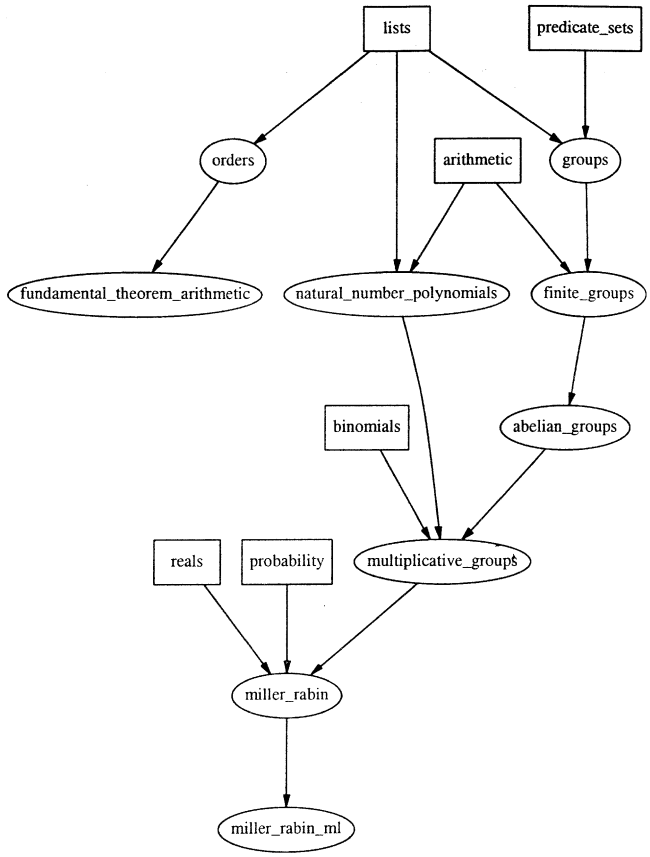


Fig. 1. The dependency relation between the theories of the HOL formalization. Boxes indicate pre-existing HOL theories, and circles are theories created for this development.

Hence by the Chinese remainder theorem $w^{2^j s} \bmod n$ cannot be equal to either 1 or $n-1$, so $w \notin B$ and the proof is complete. □

### 2.3. Formalization

Formalizing this proof in HOL was a long but mostly routine task, resulting in the theory hierarchy depicted in Fig. 1.[6] The high-level steps in the formal proofs closely follow the textbook proofs, but inevitably the bulk of the work was the low-level manipulation required to bridge between high-level steps or finish off subgoals that the textbooks regarded as sufficiently obvious as to not require any more explanation.

As we saw in the previous section, the informal proof depends heavily on groups, and the most time-consuming activity of the formalization was a thorough development of group theory. Starting from the initial axioms, we created several HOL theories of to support the classical results that were necessary for the proof, including: Lagrange's theo-

[6] The underlying probability theory in Fig. 1 is described in an earlier publication [11], and is also included in the latest release of the HOL theorem prover.

rem (9), Fermat's little theorem for groups (10) and the structure theorem for Abelian groups (11):

$$\vdash \forall\, G \in \mathsf{finite\_group}.\ \forall\, H \in \mathsf{subgroup}\ G.\ |\mathsf{set}\ H|\ \big|\ |\mathsf{set}\ G| \tag{9}$$

$$\vdash \forall\, G \in \mathsf{finite\_group}.\ \forall\, g \in \mathsf{set}\ G.\ g^{|\mathsf{set}\ G|} = e \tag{10}$$

$$\vdash \forall\, G \in \mathsf{finite\_group}. \tag{11}$$
$$\mathsf{abelian}\ G \Rightarrow \exists\, g \in \mathsf{set}\ G.\ \forall\, h \in \mathsf{set}\ G.\ h^{|g|} = e$$

This development also allowed some classical arithmetic theorems to be rendered in the language of groups, including the Chinese remainder theorem (12) and the result that the multiplicative group for a prime power is cyclic (13):

$$\vdash \forall\, p, q. \tag{12}$$
$$1 < p\ \wedge\ 1 < q\ \wedge\ \mathsf{gcd}\ p\ q = 1\ \Rightarrow$$
$$(\lambda\, x.\ (x \bmod p, x \bmod q)) \in$$
$$\mathsf{group\_iso}\ (\mathsf{mult\_group}\ pq)$$
$$(\mathsf{prod\_group}\ (\mathsf{mult\_group}\ p)\ (\mathsf{mult\_group}\ q))$$

$$\vdash \forall\, p, a. \tag{13}$$
$$\mathsf{odd}\ p\ \wedge\ \mathsf{prime}\ p\ \wedge\ 0 < a\ \Rightarrow\ \mathsf{cyclic}\ (\mathsf{mult\_group}\ p^a)$$

As well as making the arithmetic theorems more concise, this rendering also allowed the main proof to proceed entirely in the language of groups, eliminating the burden of switching mathematical context in the middle of a mechanical proof and incidentally mirroring the informal proof in Section 2.2.

The most difficult part of the whole formalization was theorem (13), guaranteeing that the multiplicative group of a prime power $p^a$ is cyclic. This proceeds by induction on $a$, and required creating whole new theories of natural number polynomials and Abelian groups for the $a = 1$ case, and a subtle argument from Baker [1] for the step case.

One surprising difference between the informal mathematics and the formalization involved the use of the fundamental theorem of arithmetic. This states that every natural number can be uniquely factorized into primes, and many informal mathematical proofs begin by applying this to some variable mentioned in the goal (e.g., by saying "let $p_1^{a_1} \cdots p_k^{a_k}$ be the prime factorization of $n$"). However, although we had previously formalized the fundamental theorem and it was ready to be applied, in the mechanical proofs we always chose what seemed to be an easier proof direction and so never needed it. Two examples of this phenomenom occur in the structure theorem for Abelian groups (11), and the cardinality of the witness set (3): the former theorem we formalized using least common multiples which proves the goal more directly; and in the latter all we needed was a case split between $n$ being a prime power or being a product of coprime $p, q$, so we separately proved this lemma.

Finally, this development provided a testing ground for a new proof tactic described in a recent paper [10]. The tactic simulates predicate subtypes in higher-order logic, taking a term $t$ and deriving particular sets $S$ such that $t \in S$ can be proved in the current logical context. It works by first recursively deriving sets for the subterms $t_i$ of $t$, and then using this information to derive sets for the term $t$. For example, consider the set $P = \{m \in \mathbb{Z} : 0 < m\}$ of positive integers, where the following theorems allow membership $t \in P$ to be deduced from knowledge of $t_i \in P$:

$$\vdash t_1 \in P \ \wedge \ t_2 \in P \ \Rightarrow \ t_1 + t_2 \in P$$
$$\vdash t_1 \in P \ \wedge \ t_2 \in P \ \Rightarrow \ t_1 t_2 \in P$$

So, for instance,

$$m \in P \ \wedge \ n \in P \ \wedge \ p \in P \ \Rightarrow \ m + mn + mnp \in P$$

The set membership $t \in P$ is clearly equivalent to the property $0 < t$, and there are many more properties that can be phrased more or less directly as set memberships. These include group membership (e.g., $g *_G h \in \mathsf{set}\ G$) and nonemptiness properties of lists and sets (e.g., $l \neq [\,]$). The predicate subtype tactic can be used to robustly prove all of these simple properties, which come up time and again as side-conditions that must be proved during term rewriting. As a consequence, this new automatic proof procedure lent itself to more efficient development of the theories that needed to be formalized in this verification, particularly the group theory where almost every theorem has one or more group membership side-conditions.

If the predicate subtype tactic had not been available, it would have been possible to use a first-order prover to show most of the side-conditions, but there are three reasons why this would have been a less attractive proposition: firstly it would have required effort to choose the right 'property propagation' theorems needed for the each goal; secondly the explicit invocations would have led to more complicated tactics; and thirdly some of the goals that can be proved using our specialized tool would simply have been out of range of a more general first-order prover.

## 3. Probability theory

### *3.1. Modelling probabilistic programs in higher-order logic*

In earlier work [11], we laid out a framework for modelling probabilistic programs in higher-order logic, in which we can specify and verify any program equipped with a source of random bits. In the language of probability theory, these random bits are assumed to be IID Bernoulli(1/2) random variables, as defined in De Groot [7, page 145]. [7] Suppose we have a probabilistic 'function'

$$\hat{f} : \alpha \to \beta$$

with an associated (deterministic) specification $B : \alpha \times \beta \to \mathbb{B}$, where a particular function application $\hat{f}(a)$ satisfies the specification if $B(a, \hat{f}(a)) = \top$. Of course, since $\hat{f}$ is probabilistic, the application $\hat{f}(a)$ may meet the specification on one instance and fail it on another.

We model $\hat{f}$ with a higher-order logic function

$$f : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$$

which explicitly takes as input a sequence $s : \mathbb{B}^\infty$ of random bits in addition to an argument $a : \alpha$, uses some of the random bits in a calculation exactly mirroring $\hat{f}(a)$, and then passes back a sequence of 'unused' bits with the result.

[7] As is common, IID is introduced as a property of a *finite* collection of random variables. We extend this to our infinite sequence by insisting that every initial segment is IID.

Mathematical measure theory is then used to define a probability measure function

$$\mathbb{P}: \mathscr{P}(\mathbb{B}^\infty) \to [0, 1]$$

from sets of sequences to real numbers between 0 and 1. The natural question "for a given $a$, with what probability does $\hat{f}$ satisfy the specification?" then becomes "for a given $a$, what is the probability measure of the set of sequences $\{s: B(a, \mathsf{fst}\ (f\ a\ s))\}$?"

This model of probabilistic functions in higher-order logic has a number of advantages:

- probabilistic programs can be represented as higher-order logic functions; we do not have to formalize another programming language in which to express the programs;
- since our modelling of probabilistic functions is the same as that used in pure functional programming languages, we can both borrow their monadic notation [21] to elegantly express our programs, and easily transfer programs to and from an execution environment;
- when applying the theory to the verification of probabilistic programs, instead of catering for myriad probability spaces we need only concern ourselves with one: sequences of IID Bernoulli$(1/2)$ bits.

To understand the probabilistic programs that follow, we now introduce the operators of the state-transformer monad. This notation allows us to reduce clutter by define probabilistic programs without directly referring to the underlying sequence of random bits; instead the unit and bind operators pass it around behind the scenes.

**Definition 4.** The state-transformer monadic operators unit and bind.

$$\vdash \forall a, s.\ \mathsf{unit}\ a\ s = (a, s)$$

$$\vdash \forall f, g, s.\ \mathsf{bind}\ f\ g\ s = \mathsf{let}\ (x, s') \leftarrow f(s)\ \mathsf{in}\ g\ x\ s'$$

The unit operator is used to lift values to the monad, and bind is the monadic analogue of function application.

### *3.2. Guaranteeing the termination of the Miller–Rabin test*

In most algorithm textbooks this is how the Miller–Rabin test is defined:

> Given an odd integer $n$ greater than 1, we pick a base $a$ at random from the set $\{1, \ldots, n-1\}$ and call witness $n\ a$. Suppose $n$ is composite: since at least $(n-1)/2$ of the bases in the set are guaranteed to be witnesses, the probability that the procedure errs is at most $((n-1)/2)/(n-1) = 1/2$.

As mentioned in the introduction, this abstract version requires a generator of uniform random numbers, but most operating systems provide only a generator of random bits. Furthermore, we have previously shown [11] that a terminating [8] algorithm to generate uniform random numbers in the range $\{0, \ldots, n-1\}$ from random bits does not exist unless $n$ is a power of 2. We therefore cannot directly implement this version of the Miller–Rabin test that is guaranteed to terminate, and it seems that we shall have to settle for termination with probability 1. However, a single observation allows the textbook algorithm to be

[8] Here we are referring to guaranteed termination on every input sequence, not termination with probability 1.

tweaked slightly, so that it satisfies the same probabilistic specification and additionally is guaranteed to terminate.

The observation is that the base 1 is always going to be a nonwitness for every $n$, so to find witnesses we can pick bases from the subset $\{2, \ldots, n-1\}$. Now if we can guarantee that the probability of picking each element from this subset is at least $1/(n-1)$, then the probability that we pick a witness is still at least $(1/(n-1))((n-1)/2) = 1/2$.

Using this observation relaxes the requirement for perfectly uniform random numbers, allowing any distribution that satisfies the lower bound. However, it is possible [11] to generate arbitrarily close approximations to uniform random numbers from random bits. This was done by introducing an extra parameter $t$, allowing us to prove the following theorem about the terminating algorithm uniform:

$$\vdash\ \forall t, n, k.\ k < n\ \Rightarrow \left|\mathbb{P}\,\{s : \mathsf{fst}\ (\mathsf{uniform}\ t\ n\ s) = k\} - 1/n\right| \leqslant 2^{-t} \tag{14}$$

Thus if we use the natural number function log2 that is related to calculating logarithms to the base 2

$$\vdash \forall n.\ \mathsf{log2}\ n = \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{suc}\ (\mathsf{log2}\ (n\ \mathsf{div}\ 2)) \tag{15}$$

$$\vdash \forall n, t.\ 0 < n\ \wedge\ 2(\mathsf{log2}\ (n+1)) \leqslant t\ \Rightarrow\ 2^{-t} \leqslant 1/n - 1/(n+1) \tag{16}$$

then the following theorem holds:

$$\begin{aligned}&\vdash \forall t, n, k. \\ &\quad k < n\ \wedge\ 2(\mathsf{log2}\ (n+1)) \leqslant t\ \Rightarrow \\ &\quad 1/(n+1) \leqslant \mathbb{P}\,\{s : \mathsf{fst}\ (\mathsf{uniform}\ t\ n\ s) = k\}\end{aligned} \tag{17}$$

Therefore, if we set the threshold $t$ to be at least $2(\mathsf{log2}(n+1))$, then for each $k \in \{0, \ldots, n-1\}$ the probability that uniform $t\ n\ s$ yields the result $k$ is at least $1/(n+1)$. Coupled with the observation above, this approximation to the uniform distribution is sufficient to implement a version of Miller–Rabin that is guaranteed to terminate.

### *3.3. Implementing the Miller–Rabin probabilistic primality test*

Having established that guaranteed termination is possible, we are now in a position to define (one iteration of) the Miller–Rabin probabilistic primality test.

**Definition 5.** A single iteration of the Miller–Rabin primality test

$$\begin{aligned}&\vdash \forall n. \\ &\quad \mathsf{miller\_rabin\_1}\ n = \\ &\quad \mathsf{if}\ n = 2\ \mathsf{then\ unit}\ \top \\ &\quad \mathsf{else\ if}\ n = 1\ \vee\ \mathsf{even}\ n\ \mathsf{then\ unit}\ \bot \\ &\quad \mathsf{else} \\ &\qquad \mathsf{bind}\ (\mathsf{uniform}\ (2(\mathsf{log2}\ (n-1)))\ (n-2)) \\ &\qquad (\lambda a.\ \mathsf{unit}\ (\neg\mathsf{witness}\ n\ (a+2)))\end{aligned} \tag{18}$$

This satisfies the correctness theorems

$$\vdash \forall n, s.\ \mathsf{prime}\ n\ \Rightarrow\ \mathsf{fst}\ (\mathsf{miller\_rabin\_1}\ n\ s) = \top \tag{19}$$

$$\vdash \forall n.\ \neg\mathsf{prime}\ n\ \Rightarrow\ 1/2 \leqslant \mathbb{P}\{s : \mathsf{fst}\ (\mathsf{miller\_rabin\_1}\ n\ s) = \bot\} \tag{20}$$

In order to define the full Miller–Rabin test using several bases, we create a new (state-transformer) monadic operator many. The intention of many $p\ n$ is a test that repeats $n$ times the test $p$ using different parts of the random bit stream, returning true if and only if each evaluation of $p$ returned true. For instance, sdest is the destructor function for a stream, and so the function many sdest 10 tests that the next 10 booleans in the random stream are all $\top$. Here is the definition of many and basic properties:

$$\begin{aligned}&\vdash \forall f, n. \\ &\quad \mathsf{many}\ f\ 0 = \mathsf{unit}\ \top\ \wedge \\ &\quad \mathsf{many}\ f\ (\mathsf{suc}\ n) = \mathsf{bind}\ f\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then\ many}\ f\ n\ \mathsf{else\ unit}\ \bot)\end{aligned} \tag{21}$$

$$\vdash \forall f, n.\ \mathbb{P}\{s : \mathsf{fst}\ (\mathsf{many}\ f\ n\ s)\} = (\mathbb{P}\{s : \mathsf{fst}\ (f\ s)\})^n \tag{22}$$

Using the new many monadic operator it is simple to define the Miller–Rabin function miller_rabin

$$\vdash \forall n, t.\ \mathsf{miller\_rabin}\ n\ t = \mathsf{many}\ (\mathsf{miller\_rabin\_1}\ n)t \tag{23}$$

and finally Theorem 1 from the introduction follows from (19)–(23).

### *3.4. A compositeness prover*

For any input prime $p$, Theorem 1 gives us a very strong guarantee: for any $t$ and any sequence $s$, our implementation of the Miller–Rabin test will always output $\top$.[9] Therefore, given a number $n$ and a sequence $s$, if miller_rabin $n$ 1 $s$ can be rewritten to $\bot$, then we can immediately deduce the higher-order logic theorem $\vdash \neg\mathsf{prime}\ n$. Such rewriting of programs in the logic can take place using either the standard HOL rewriting tools, or the `computeLib` tool of Barras [2] which is guaranteed to match the complexity of execution in ML.[10]

All that is required is an input sequence containing the 'random' bits that the Miller–Rabin test will use in its execution. This is easily created by defining a pseudo-random bit generator in the logic, and for this we just need an initial seed $d{:}\alpha$ and an iteration function $i : \alpha \to \mathbb{B} \times \alpha$. For example, we can adapt to our purpose the linear congruence method of generating pseudo-random numbers [13]:

$$\begin{aligned}d &= 0 \\ i &= \lambda n.\ (\mathsf{even}\ n,\ An + B \bmod (2N + 1))\end{aligned}$$

[9] Such a strong result is a consequence of guaranteed termination (as opposed to termination with probability 1), which is why in Section 3.2 we took such care to ensure this.

[10] Barras calculates the constant factor difference to be about 1000 for his merge-sort example.

Table 1
Testing the composite prover

| $n$ | Run time | GC time |
|---|---|---|
| $2^{2^5}+1$ | 53.080 | 7.170 |
| $2^{2^6}+1$ | 370.210 | 53.530 |
| $2^{2^7}+1$ | 2842.920 | 409.620 |
| $2^{2^8}+1$ | 22095.770 | 3170.780 |

We emphasize that this family of sequences is merely a convenient way to generate superficially unpredictable bits, and is of course completely deterministic. [11] The point is well made by von Neumann [20]:

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number—there are only methods to produce random numbers, and a strict arithmetical procedure is of course not such a method.

For our purpose, we set the parameters to be $A = 103$, $B = 95$, $N = 79$, and we now have all the ingredients we need to execute the Miller–Rabin primality test in the logic using `computeLib`. Table 1 gives the run times in seconds (including garbage collection (GC) time) that was taken to prove the compositeness of some Fermat numbers. [12]

The last number in Table 1 has 78 digits, and demonstrates the possibility of formally proving numbers to be composite with no knowledge of their factors. Since this proof procedure is not the main focus of this paper (rather an interesting digression), we shall not do any more profiling than this. However, one point that can be made from the existing results is that from one line of the table to the next, the number of digits in $n$ roughly doubles while the run time increases by a factor of 8. This cubic growth is indeed what we would expect if we executed the algorithm in ML, empirically confirming the theoretical result that the efficiency of `computeLib` is a constant factor away from ML.

## 4. Extracting the algorithm to standard ML

The advantage of extracting the algorithm to a standard programming language such as ML is twofold: firstly execution is more efficient, and so the algorithm can be applied to usefully large numbers; and secondly it can be packaged up as a module and used as a reliable component of larger programs.

However, there is a danger that the properties that have been verified in the theorem-prover are no longer true in the new context. In this section we make a detailed examination of the following places where the change in context might potentially lead to problems: the source of random bits, the arbitrarily large natural numbers, and the manual translation of the Miller–Rabin functions to ML. Finally we test the algorithm on some examples, to check again that nothing has gone amiss and also to get some idea of the performance and computational complexity of the code.

[11] We might equally well test our programs on the sequence where every element is $\top$, but this is not even superficially unpredictable.

[12] All the results in this paper were produced using the Moscow ML 2.00 interpreter over RedHat Linux 6.2, running on a computer with a 200 MHz Pentium Pro processor and 128 MB of RAM.

### 4.1. Random bits

Our theorems are founded on the assumption that our algorithms have access to a generator of perfectly random bits: each bit has probability of exactly $1/2$ of being either 1 or 0, and is completely independent of every other bit. In the real world this idealized generator cannot exist, and we must necessarily select an approximation.

The first idea that might be considered is to use a pseudo-random number generator, such as the linear congruence method we used to execute the Miller–Rabin test in the logic. These have been extensively analysed (for example by Knuth [13]) and pass many statistical tests for randomness, but their determinism makes them unsuitable for applications that require genuine unpredictability. For instance, when generating cryptographic keys it is not sufficient that the bits appear random, they must be truly unpredictable even by an adversary intent on exploiting the random number generator used.

Rejecting determinism, we must turn to the operating system for help. Many modern operating systems can utilize genuine non-determinism in the hardware to provide a higher quality of random bits. For example, here is a description of how random bits are derived and made available in Linux, excerpted from the `man` page '`random`' in Section 4:

> The random number generator gathers environmental noise from device drivers and other sources into an entropy pool. The generator also keeps an estimate of the number of bit[s] of the noise in the entropy pool. From this entropy pool random numbers are created.
>
> When read, the `/dev/random` device will only return random bytes within the estimated number of bits of noise in the entropy pool. `/dev/random` should be suitable for uses that need very high quality randomness such as one-time pad or key generation. When the entropy pool is empty, reads to `/dev/random` will block until additional environmental noise is gathered.
>
> When read, `/dev/urandom` device will return as many bytes as are requested. As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver. Knowledge of how to do this is not available in the current non-classified literature, but it is theoretically possible that such an attack may exist. If this is a concern in your application, use `/dev/random` instead.

These devices represent the highest quality source of randomness to which we have easy access, and so we have packaged them up as ML boolean streams for use in our extracted program.

### 4.2. Arbitrarily large natural numbers

Another place where there is a potential disparity between HOL and ML regards their treatment of numbers. The HOL Miller–Rabin test operates on the natural numbers {0, 1, 2, ...}, while in ML the primitive type `int` contains signed numbers in a range depending on the machine architecture.

We resolved this incompatibility by creating the ML module `HolNum`, which implements an equality type `num` of arbitrarily large natural numbers. The Miller–Rabin functions may then use this type of numbers, and the arithmetic operations will behave exactly as in HOL.

We first implemented our own large number module, written in the purely functional subset of ML (and using a `word vector` representation of numbers). However, this was

found to be about 100 times slower than the Moscow ML interface to the GNU Multi-Precision library, so we switched to this instead.

### 4.3. Extracting from HOL to ML

A further place where errors could creep in is the manual extraction of the Miller–Rabin functions from HOL to ML. Consequently we did this in two steps, the first of which was creating a new HOL theory containing a version of each function we wished to extract. For example, in Theorem 21 the monadic operator many was defined like so

$$\begin{aligned}&\vdash \forall\, f, n.\\ &\quad \mathsf{many}\ f\ 0 = \mathsf{unit}\ \top\ \wedge\\ &\quad \mathsf{many}\ f\ (\mathsf{suc}\ n) = \mathsf{bind}\ f\ (\lambda\, x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{many}\ f\ n\ \mathsf{else}\ \mathsf{unit}\ \bot)\end{aligned}$$

and in this new HOL theory we prove it is equivalent to

$$\begin{aligned}&\vdash \forall\, f, n.\\ &\quad \mathsf{many}\ f\ n =\\ &\quad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{unit}\ \top\\ &\quad \mathsf{else}\ \mathsf{bind}\ f\ (\lambda\, x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{many}\ f\ (n-1)\ \mathsf{else}\ \mathsf{unit}\ \bot)\end{aligned}$$

so that we may export it to ML as

```
fun MANY f n =
  if n = ZE\-RO then UNIT true
  else BIND f (fn x => if x then MANY f (n -- ONE) else UNIT false);
```

As can be seen here the ML version involves some lexical changes, but has precisely the same parse tree as the intermediate HOL version. This reduces the chance of errors introduced by the cut-and-paste operation.

An intellectually interesting problem in the extraction is the question of how to handle partial functions. Consider the HOL function uniform that generates (approximations to) uniform random numbers:

$$\vdash \forall\, t, n.\ \mathsf{uniform}\ t\ (\mathsf{suc}\ n) = \mathsf{if}\ t = 0\ \mathsf{then}\ \mathsf{unit}\ 0\ \mathsf{else}\ \ldots$$

The function is deliberately underspecified: there is no case where the second argument takes the value 0 because it does not make sense to talk of random numbers uniformly distributed over the empty set. HOL allows us to define functions like this, but there is no immediate ML equivalent. In the intermediate HOL version, we prove it to be equivalent to

$$\begin{aligned}&\vdash \forall\, t, n.\\ &\quad \mathsf{uniform}\ t\ n =\\ &\quad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{uniform}\ t\ n\ \mathsf{else}\ \mathsf{if}\ t = 0\ \mathsf{then}\ \mathsf{unit}\ 0\ \mathsf{else}\ \ldots\end{aligned}$$

This rather strange-looking theorem is a formulation of uniform that is suitable to be extracted to ML, because the left hand side does not contain any illegal patterns like $\mathsf{suc}\ n$.

Of course, if we naively extract this to ML it will loop forever when a user mistakenly calls it with second argument 0, so we improve the error handling by extracting in the following way:

```
fun uni\-form t n =
  if n = ZE\-RO then raise Fail "uni\-form: out of do\-main"
  else if t = ZE\-RO then UNIT ZE\-RO
  else ...
```

This implementation trick allows us to faithfully extract partial functions with as much confidence as for total functions.

Finally, we note that had we completed our verification in the Coq theorem prover, then the standard Coq infrastructure could have been used to extract the same verified Miller–Rabin test to ML. In addition, our stricving for guaranteed termination also allows a relatively straightforward constructive proof, since only rational numbers can appear as probabilities. However, similar experience has shown that the constructive proof would still be harder work than our classical one, and the restriction to guaranteed termination rules out the straightforward verification of many interesting programs that termination with probability 1.

### *4.4. Testing*

It would be pleasant to say that since the function had been mechanically verified, no testing was necessary. But the preceding subsections have shown that this would be naive. Even if we are prepared to trust the generation of random bits, the operations of our arbitrarily large number module and the manual extraction of the algorithms, testing would still be prudent to catch bugs at the interface between these components.

The first quick test was an ML version of Rabin's $2^{400}$ test mentioned in the introduction: with the number of tries (the $t$ parameter) set to 50 the program took 15 s to confirm that $2^{400} - 593$ is indeed the smallest (probable) prime below $2^{400}$.

The main test proceeded in the following way: for various values of $l$, generate $n$ odd candidate numbers of length $l$ bits. Perform a quick compositeness test on each by checking for divisibility by the first $l$ primes, and also run Miller–Rabin with the maximum number of bases fixed at 50. The results are displayed in Table 2. $\mathbb{E}_{l,n}(\text{composite})$ is equal to $n(1 - \mathbb{P}_l(\text{prime}))$ and mathematically estimates the number of composites that the above algorithm will consider as candidates. [13]

QC is the number of candidates that were found to be divisible by the quick compositeness test, MR is the number that the Miller–Rabin algorithm found to be composite, and finally $\text{MR}_+$ is the number of candidates that the Miller–Rabin algorithm needed more than one iteration to determine that it was composite.

The most important property for testing purposes cannot be deduced from the table: for each number that the quick compositeness test found to be composite, the Miller–Rabin

[13] Using the prime number theorem, $\pi(n) \sim n/\log n$, we can derive:

$$\mathbb{P}_l(\text{prime}) = \frac{\pi(2^l) - \pi(2^{l-1})}{2^{l-2}} \sim \frac{2}{\log 2}\left(\frac{2}{l} - \frac{1}{l-1}\right).$$

Table 2
Testing the extracted Miller–Rabin algorithm

| $l$ | $n$ | $\mathbb{E}_{l,n}$ | QC | MR | $\mathrm{MR}_+$ |
|---|---|---|---|---|---|
| 10 | 100,000 | 74,352 | 70,262 | 70,683 | 520 |
| 15 | 100,000 | 82,138 | 72,438 | 80,448 | 85 |
| 20 | 100,000 | 86,332 | 74,311 | 85,338 | 5 |
| 50 | 100,000 | 94,347 | 79,480 | 94,172 | 0 |
| 100 | 100,000 | 97,144 | 82,258 | 97,134 | 0 |
| 150 | 100,000 | 98,089 | 83,401 | 98,077 | 0 |
| 200 | 100,000 | 98,565 | 84,370 | 98,557 | 0 |
| 500 | 100,000 | 99,424 | 86,262 | 99,458 | 0 |
| 1000 | 100,000 | 99,712 | 87,377 | 99,716 | 0 |
| 1500 | 100,000 | 99,808 | 87,935 | 99,798 | 0 |
| 2000 | 100,000 | 99,856 | 88,342 | 99,852 | 0 |

Table 3
Profiling the extracted Miller–Rabin algorithm

| $l$ | Gen time | QC time | $\mathrm{MR}_1$ time |
|---|---|---|---|
| 10 | 0.0004 | 0.0014 | 0.0028 |
| 15 | 0.0007 | 0.0017 | 0.0041 |
| 20 | 0.0009 | 0.0019 | 0.0054 |
| 50 | 0.0023 | 0.0034 | 0.0136 |
| 100 | 0.0068 | 0.0075 | 0.0370 |
| 150 | 0.0107 | 0.0112 | 0.0584 |
| 200 | 0.0157 | 0.0156 | 0.0844 |
| 500 | 0.0443 | 0.0416 | 0.2498 |
| 1000 | 0.0881 | 0.0976 | 0.7284 |
| 1500 | 0.1543 | 0.2164 | 1.7691 |
| 2000 | 0.3999 | 0.2843 | 4.2910 |

test also returned this result (and as can be seen, this almost always required only one iteration). In the other direction, using the $\mathbb{E}_{l,n}$ column as a guide, we can see that the Miller–Rabin algorithm did not find many more composites than expected.

In Table 3 we compare for each $l$ the average time in seconds taken to generate a random odd number, subject it to the quick composite test, and perform one iteration of the Miller–Rabin algorithm.

The complexity of (one iteration of) the Miller–Rabin algorithm is around $O(l^2 \log l)$, since it uses asymptotically the same number of operations as modular exponentiation [6]. However, performing linear regression on the log–log graph in Fig. 2 gives a good fit with degree 1.32, implying a complexity of $O(l^{1.32})$. We can only conclude that the GNU Multi-Precision library is heavily optimized for the 'small' numbers in the range we were using, and so we cannot expect an asymptotically valid result. [14]

[14] When we ran this experiment using our own purely functional implementation of arbitrarily large numbers, it was a different story. Performing linear regression on the log–log graph gave a good fit with degree 2.98, confirming the theoretical result since we used the simple $O(l^2)$ algorithm for multiplication. GC was minimal, typically accounting for less than 5% of the time taken.

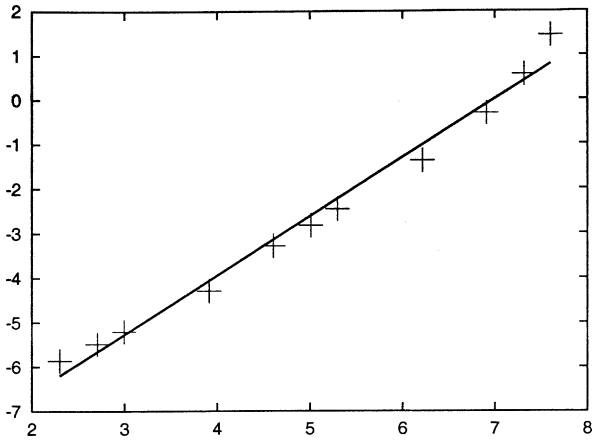


Fig. 2. Graph of $\log(\mathrm{MR}_1$ time) against $\log l$.

## 5. Conclusion

In this paper we have shown that our higher-order logic framework for verifying probabilistic programs is powerful enough to formally specify and verify the Miller–Rabin primality test, a well-known and commercially used probabilistic algorithm. The verification highlighted a small gap between theory and implementation, namely the difference between having a generator of uniformly distributed random numbers and a generator of random bits. An extra observation bridged this gap in the proof, and we were able to produce a version of Miller–Rabin using random bits that was guaranteed to terminate and satisfied the required probabilistic specification. An immediate application was a procedure for formally proving in HOL the compositeness of a number, without requiring a witness factor.

The predicate set prover helped to make the proof development more efficient; it was particularly useful for proving group membership conditions and simple but ubiquitous arithmetic conditions. Our evaluation is that it is a useful tool for reasoning about term properties that naturally propagate up terms, and a useful condition prover for contextual rewriters.

The difference between formal and informal proofs in their use of the fundamental theorem of arithmetic was pointed out in Section 2.3. This is the most striking example of many small differences in the style of informal and formal proofs, stemming from the different proof consumers in each case. Machines make it easier to formalize principles of induction such as dividing out a prime or prime power factor of a number, whereas humans would seem to be better at manipulating the multisets that contain the prime factors. However, it is usually straightforward to translate standard proofs from the literature into the form that is simpler to machine check.

We also extracted the algorithm to standard ML that takes as input a number and a stream of random bits, and declares the number either to be composite or probably prime, with a formally specified probability. Algorithms such as these with a probabilistic specification are difficult to get right, since testing must necessarily be statistical. In this paper we have given arguments that our version has a high assurance of correctness.

Overall, the proof script for the whole verification is 8000 lines long over 16 theories. It is difficult to measure the time it took to perform the verification, since at the same time

there was much development of the HOL probability theory. Perhaps six weeks would be a reasonable estimate, mainly due to the formalization of number theory results. Of course, the formalized number theory is now available to use for any user of the HOL theorem prover, and the framework for verifying probabilistic programs has already been applied to several other examples [11]. However, the Miller–Rabin primality test remains our largest verification to date.

## 6. Related Work

There has been a long history of number theory formalizations, most relevantly for us: Russinoff's proof of Wilson's theorem in the Boyer–Moore theorem-prover [17]; Boyer and Moore's correctness proof of the RSA algorithm in ACL2 [4]; and Théry's correctness proof of RSA in three different theorem-provers [19]. This last work was especially useful, since one of the theorem-provers was HOL, and we were able to use his proof of the binomial theorem in our own development.

The closest work in spirit to this paper is Caprotti and Oostdijk's [5] primality proving in Coq, in which they formalize a similar computational number theory development and utilize a computer algebra system to prove numbers prime. Seeing this work improved the organization of theories in our own formalism. Harrison has also implemented a primality prover in HOL Light, using Pratt's criterion instead of Pocklington's.

Our own development of group theory benefitted from the higher-order logic formalisms of Gunter [8], Kammüller [12] and Zammit [22], but the theory of groups has been formalized in many different theorem-provers.

There exist other formalizations of probability, and some have been applied to analysing probabilistic programs. It is conceivable that the same work could have been carried out using the probabilistic predicate transformers of Morgan et al. [15], except that this formalism has not yet been mechanized.

## Acknowledgements

This paper represents the fruits of a long-term project, and I would like to thank Judita Preiss, Michael Norrish, Konrad Slind, and my supervisor Mike Gordon for their encouragement and advice throughout. Also David Preiss and Desmond Sheiham helped me to interpret some tricky textbook mathematics proofs; without their help the formalization of computational number theory would not have been so smooth. Finally, comments from the JLAP referees helped to improve many aspects of the paper.

## References

[1] A. Baker, A Concise Introduction to the Theory of Numbers, Cambridge University Press, 1984.
[2] B. Barras, Programming and computing in HOL, in: M. Aagaard, J. Harrison (Eds.), Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000, vol. 1869 of Lecture Notes in Computer Science, Portland, OR, USA, August 2000, Springer, pp. 17–37.
[3] E.R. Berlekamp, Factoring polynomials over large finite fields, Mathematics of Computation 24 (1970).
[4] R.S. Boyer, J.S. Moore, Proof checking the RSA public key encryption algorithm, American Mathematical Monthly 91 (3) (1984) 181–189.

[5] O. Caprotti, M. Oostdijk, Formal and efficient primality proofs by use of computer algebra oracles, Journal of Symbolic Computation 32 (1–2) (2001) 55–70 (Special issue on Computer Algebra and Mechanized Reasoning).
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press/McGraw-Hill, Cambridge, MA, 1990.
[7] M. DeGroot, Probability and Statistics, second ed., Addison-Wesley, 1989.
[8] E. Gunter, Doing algebra in simple type theory, Technical Report MS-CIS-89-38, Logic & Computation 09, Department of Computer and Information Science, University of Pennsylvania, 1989.
[9] G.H. Hardy, A Mathematician's Apology, reprinted with a foreword by C.P. Snow, Cambridge University Press, 1993.
[10] J. Hurd, Predicate subtyping with predicate sets, in: R.J. Boulton, P.B. Jackson (Eds.), 14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001, volume 2152 of Lecture Notes in Computer Science, pages 265–280, Edinburgh, Scotland, September 2001. Springer, Berlin.
[11] J. Hurd, Formal verification of probabilistic algorithms, PhD thesis, University of Cambridge, 2002.
[12] F. Kammüller, L.C. Paulson, A formal proof of Sylow's first theorem—an experiment in abstract algebra with Isabelle HOL, Journal of Automated Reasoning 23 (3–4) (1999) 235–264.
[13] D.E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, third ed., Addison-Wesley, 1997.
[14] G.L. Miller, Riemann's hypothesis and tests for primality, in: Conference Record of Seventh Annual ACM Symposium on Theory of Computation, Albuquerque, New Mexico, May 1975, pp. 234–239.
[15] C. Morgan, A. McIver, K. Seidel, J.W. Sanders, Probabilistic predicate transformers, Technical Report TR-4-95, Oxford University Computing Laboratory Programming Research Group, February 1995.
[16] M.O. Rabin, Probabilistic algorithms, in: J.F. Traub (Ed.), Algorithms and Complexity: New Directions and Recent Results, Academic Press, New York, 1976, pp. 21–39.
[17] D.M. Russinoff, An experiment with the Boyer–Moore theorem prover: a proof of Wilson's theorem, Journal of Automated Reasoning 1 (1985) 121–139.
[18] R. Solovay, V. Strassen, A fast Monte-Carlo test for primality, SIAM Journal on Computing 6 (1) (1977) 84–85.
[19] L. Théry, A quick overview of HOL and PVS, August 1999, Lecture Notes from the Types Summer School '99: Theory and Practice of Formal Proofs, held in Giens, France.
[20] J. von Neumann, Various techniques for use in connection with random digits, in: von Neumann's Collected Works, vol. 5, Pergamon, 1963, pp. 768–770.
[21] P. Wadler, The essence of functional programming, in: 19th Symposium on Principles of Programming Languages, ACM Press, January 1992.
[22] V. Zammit, On the Readability of Machine Checkable Formal Proofs, PhD thesis, University of Kent at Canterbury, March 1999.