Under consideration for publication in Math. Struct. in Comp. Science

Randomized Algorithms for Isomorphisms of Simple Types

JOSEPH (YOSSI) GIL[†] and YOAV ZIBIN

Technion—Israel Institute of Technology

Technion City, Haifa, 32000, Israel

Email: yogi | zyoav @ cs.technion.ac.il

Received June 2005; Revised October 2006

We give the first linear time (yet randomized) algorithm for the *first order isomorphism problem*, i.e., the isomorphism of non-recursive types involving product- and function- type constructors, under the axioms of commutativity and associativity of products, currying and distributivity of functions over products. This problem can also be thought of as the problem of formal equality-testing of multi-variate expressions involving only multiplications and exponentiation. Previous work gave a deterministic $O(n \log^2 n)$ time and $O(n)$ space algorithm for the problem (n being the input size). Our specific contribution includes two randomized algorithms for the problem: (i) an $O(n)$ time *Monte Carlo* algorithm (i.e., with a small probability it may decide erroneously that the two types are isomorphic), and (ii) an $O(n \log n)$ expected time and $O(n)$ space *Las Vegas* algorithm (i.e., with a small probability it may execute long). The algorithms rely on a preprocessing stage which computes the sequence of the first n primes in $O(n \log n/ \log\log n)$ time and space.

1. Introduction

The problem of testing for formal equality of multi-variate polynomials attracted a lot of attention (see [Chen and Kao, 1997] for a recent survey) and its simple $O(n \log n)$ time (n being the input size) algorithm (see e.g., [Schwartz, 1980]) has found many applications, such as in perfect matching and in multi-set equality. Polynomials are expressions involving additions and multiplications over a set of formal variables. We are concerned with the problem variant in which the input expressions involve multiplications and exponentiations (*ME-expressions*), e.g., testing the formal equality

$$\left((\mathsf{ab})^{(\mathsf{a}^{\mathsf{b}})}\right)^{(\mathsf{b}^{\mathsf{a}})} = \mathsf{a}^{\mathsf{a}^{\mathsf{b}}\mathsf{b}^{\mathsf{a}}}\mathsf{b}^{\mathsf{b}^{\mathsf{a}}\mathsf{a}^{\mathsf{b}}}. \tag{1.1}$$

Despite the similarity in formulation, our variant is different from the original since the governing rules of algebra of ME-expressions are quite distinct than those of polynomials. Also, there is no standard normal form for presenting ME-expressions. Most importantly, subtle properties of

[†] Research supported in part by the generous funding of the Israel Science Foundation, grant No. 128/2002.

exponentiations in finite fields are a major hurdle in adapting to our variant the technique of evaluating polynomials at random points.

For a long time, the problem was thought to have an exponential nature [Soloviev, 1983, Bruce et al., 1991]. Considine [Considine, 2000] gave an $O(n^2 \log n)$ time and $O(n^2)$ space algorithm for the problem. This result was later improved [Zibin et al., 2003] to $O(n \log^2 n)$ time and $O(n)$ space.

This paper gives a linear time Monte-Carlo algorithm (i.e., with a small probability it may decide erroneously that the two types are isomorphic), and an $O(n \log n)$ time Las-Vegas algorithm (i.e., with a small probability it may execute long). Both our algorithms employ what can be thought of as a normal form for ME-expressions. Even though this normal form may require quadratic size, we show that it has a more compact linear representation with a tree-like node sharing.

Our results indicate that equality of ME-expressions is easier than that of polynomials. For completeness, it should be mentioned that if the input expressions involve additions, multiplications *and* exponentiation, then the problem escalates to being Tarski's *high school algebra problem* [Tarski, 1951], which has a non-polynomial algorithm [Gurevič, 1985], despite not having a finite set of axioms.

Beyond the theoretical interest, our attention to ME-expressions was drawn by practical applications. ME-expressions conveniently encode *first-order types* which are used extensively in functional programming languages: primitive types are encoded as formal variables, record types as multiplication, and functions as exponentiation. For example, a function receiving an **`integer`** and returning a **`real`** is encoded as the expression r^{i}, where i is the encoding of **`integer`** and r is the encoding of **`real`**. The equality $(\mathsf{a} \times \mathsf{b})^{\mathsf{c}} = \mathsf{a}^{\mathsf{c}} \times \mathsf{b}^{\mathsf{c}}$ means that a function receiving an argument of type c and returning a pair of values of types a and b, is isomorphic to a pair of functions, one from c to a and the other from c to b. Similarly, $\mathsf{a}^{\mathsf{bc}} = \left(\mathsf{a}^{\mathsf{b}}\right)^{\mathsf{c}}$ means that a function taking two arguments of types b and c and returning a value of type a, is isomorphic to a function receiving a c and returning yet another function which takes a b and returns an a. We shall therefore refer to the problem of formal equality of ME-expressions, as the *first-order type isomorphism problem*.

Algorithms for type isomorphism are useful in searching large software libraries [Rittri, 1990, Di Cosmo, 1995], where the desired type of a function is used as a search key and functions with isomorphic types are returned as candidates. We even note that our Las-Vegas algorithm produces a proof of isomorphism between the input types, which can generate bridge code converting the functions found to the desired type.

Outline Pertinent definitions are made in Section 2. Section 3 defines the function $g : \mathcal{E} \mapsto \mathbb{N}$ which is a Gödel-like encoding of ME-expressions which satisfies $\epsilon_1 = \epsilon_2 \Leftrightarrow g(\epsilon_1) = g(\epsilon_2)$. Section 4 describes the Monte Carlo algorithm, while Section 5 transforms it into a Las Vegas algorithm. Finally, Section 6 concludes and describes some open problems.

2. Definitions

Let $\mathcal{X}$ be a set of symbols, which can be thought as primitive types, or variables. Then, the set $\mathcal{E}$ of *ME-expressions* (or for short *types* or *expressions*) over $\mathcal{X}$ is defined by the following abstract grammar

$$e ::= \mathbf{1} \quad | \quad x \quad | \quad e \times e \quad | \quad e^e.$$

Given two expressions $\epsilon_1, \epsilon_2 \in \mathcal{E}$, the problem at the focus of our attention is the decision whether they are "equal". The problem size $n = |\epsilon_1| + |\epsilon_2|$ is the sum of sizes of the two input expressions, with the following slightly non-standard definition of expression size.

Definition 2.1 (Size of expression). The *size* of an expression $e \in \mathcal{E}$, denoted $|e|$, is defined as

$$|e| = \begin{cases} 1 + |e_1| + |e_2| & \text{if } e = e_1 \times e_2 \\ 2 + |e_1| + |e_2| & \text{if } e = e_1^{e_2} \\ 2 & \text{if } e = x \\ 1 & \text{if } e = \mathbf{1} \end{cases}$$

(The reasons for picking this particular definition will become apparent in the proof of Lemma 3.4 below. Note that $|e|$ is at most twice the number of nodes in the tree representation of e.)

We also need a precise definition of equality. Equality is obviously symmetric and transitive, and as usual, equality is preserved by composition. In other words, the theory of equality includes the following four inference rules:

$$\frac{A = B}{B = A} \qquad \text{Symmetry}$$

$$\frac{A = B, C = D}{A \times C = B \times D} \qquad \text{Congruence of } \times$$

$$\frac{A = B, B = C}{A = C} \qquad \text{Transitivity}$$

$$\frac{A = B, C = D}{A^C = B^D} \qquad \text{Congruence of exponentiation}$$

The following eight axioms represent simple arithmetical rules, and are also pertinent to the definition of equality:

$(\mathcal{A}.0)$	$A = A$	(Reflexivity)
$(\mathcal{A}.1)$	$A \times \mathbf{1} = A$	
$(\mathcal{A}.2)$	$\mathbf{1}^A = \mathbf{1}$	
$(\mathcal{A}.3)$	$A^{\mathbf{1}} = A$	
$(\mathcal{A}.4)$	$A \times B = B \times A$	(Commutativity)
$(\mathcal{A}.5)$	$A \times (B \times C) = (A \times B) \times C$	(Associativity)
$(\mathcal{A}.6)$	$(A^B)^C = A^{B \times C}$	(Currying)
$(\mathcal{A}.7)$	$(A \times B)^C = (A^C) \times (B^C)$	(Distributivity)

The simplest equalities are obtained by instantiating the axioms.

Definition 2.2 (Axiom instance). An *instance of an axiom* $\mathcal{A}$ is the result of a consistent substitution of all the variables in $\mathcal{A}$ by expressions from $\mathcal{E}$.

For example, $(\mathbf{1} \times \mathsf{a}^{\mathsf{b}}) \times \mathsf{c} = \mathsf{c} \times (\mathbf{1} \times \mathsf{a}^{\mathsf{b}})$ is an instance of the Commutativity axiom, rising from the substitution $\{A \mapsto (\mathbf{1} \times \mathsf{a}^{\mathsf{b}}), B \mapsto \mathsf{c}\}$.

All other equalities are obtained by the application of the inference rules in a derivation sequence.

Definition 2.3 (Derivation sequence). The sequence

$$e_1 = e_1' \quad , \quad e_2 = e_2' \quad , \quad \dots \quad , \quad e_m = e_m'$$

is called a *derivation sequence* if for $i = 1, 2, \dots, m$, $e_i = e_i'$ is either an instance of an axiom or the result of applying one of the four inference rules to previous equalities.

Definition 2.4 (Equality). Let $\epsilon_1, \epsilon_2 \in \mathcal{E}$ be two expressions. Then, $\epsilon_1 = \epsilon_2$ (read ϵ_1 is *equal* to ϵ_2) if and only if there exists a derivation sequence ending with $\epsilon_1 = \epsilon_2$.

The above axioms and inference rules were proved to be complete for the Cartesian closed categories [Bruce et al., 1991]. Therefore, $\epsilon_1 = \epsilon_2$ is true if and only if the functions ϵ_1 and ϵ_2 represent are equal in all Cartesian closed categories. Since the category of finite sets is also complete [Soloviev, 1983] we have that $\epsilon_1 = \epsilon_2$ if and only if the functions ϵ_1 and ϵ_2 are always equal over the set of natural numbers $\mathbb{N}$ (excluding zero).

We also use the following notation: The set of prime numbers is denoted $\mathbb{P} \subset \mathbb{N}$. We use the symbol $\circ$ to denote function composition, i.e., $f \circ g(x) \equiv f(g(x))$.

3. A Gödel-Like Encoding of Expressions

In this section we shall construct a function $g : \mathcal{E} \mapsto \mathbb{N}$, which encodes expressions as natural numbers. The encoding is Gödel-like in that it uses unique primes to encode the symbols in $\mathcal{X}$. However, unlike classical Gödel encodings, the function g preserves some of the structure of $\mathcal{E}$. Moreover, we shall require that $\epsilon_1 = \epsilon_2$ *if and only if* $g(\epsilon_1) = g(\epsilon_2)$. At this stage, we shall not concern ourselves with the size of the encoding, nor with many of the algorithmic details of an implementation.

If the expressions do not include an exponentiation, then an encoding of expressions which preserves equality can be obtained by a simple assignment of prime numbers to symbols, and defining $g(e_1 \times e_2) = g(e_1) \cdot g(e_2)$. However, this property is no longer true when exponentiation is allowed. In other words, if we encode symbolic exponents into integer exponentiation, $g(e_1^{e_2}) = g(e_1)^{g(e_2)}$, then we may encode non-equal expressions as equal numbers, i.e., $\epsilon_1 = \epsilon_2 \Rightarrow g(\epsilon_1) = g(\epsilon_2)$ holds, but the reverse will not necessarily hold. Consider, for example, the following two expressions:

$$\epsilon_1 = \mathsf{a}^{\mathsf{b}},$$
$$\epsilon_2 = \mathsf{a} \times \mathsf{a} \times \mathsf{a}.$$

Clearly those two expressions are not equivalent, but if we assign $g(\mathsf{a}) = 2$ and $g(\mathsf{b}) = 3$ then the encodings are equal. As another example, the encoding of the two expressions

$$\mathsf{a}^{\mathsf{b}_1} \times \mathsf{a}^{\mathsf{b}_2} \times \mathsf{a}^{\mathsf{b}_3} \times \mathsf{a}^{\mathsf{b}_4},$$
$$\mathsf{a}^{\mathsf{c}_1} \times \mathsf{a}^{\mathsf{c}_2} \times \mathsf{a}^{\mathsf{c}_3} \times \mathsf{a}^{\mathsf{c}_4},$$

will be the same if $\alpha(\mathsf{b}_1)+\alpha(\mathsf{b}_2)+\alpha(\mathsf{b}_3)+\alpha(\mathsf{b}_4) = \alpha(\mathsf{c}_1)+\alpha(\mathsf{c}_2)+\alpha(\mathsf{c}_3)+\alpha(\mathsf{c}_4)$, where $\alpha(x)$

assigns a unique prime to x. Thus, the encodings will be equal in the common case that the sum of one quadruple of primes is equal to that of another such quadruple.

Instead, we bring all exponentiations to the form x^e, $x \in \mathcal{X}$. Expressions in this form are then *encoded* (rather than computed) as a *new* unique prime. This encoding can be thought of as a generalization of assignment: The assignment function $\alpha : \mathcal{X} \times \mathbb{N} \mapsto \mathbb{P}$, is defined so that $\alpha(x, \eta)$, where $\eta = g(e)$, returns a unique prime associated with x^e. We can also write

$$g(x^e) = \alpha(x, g(e)) . \tag{3.1}$$

The special case of assignment of a unique prime to an isolated primitive x is given by $\alpha(x, 1)$ since $x = x^{\mathbf{1}}$ and $1 = g(\mathbf{1})$. We use the abbreviation $\alpha(x) = \alpha(x, 1)$.

Consider, for example, the expressions $\epsilon_1 = \mathsf{a} \times \mathsf{c}^{\mathsf{a} \times \mathsf{b}} \times \mathsf{d}^{\mathsf{b} \times \mathsf{a}}$ and $\epsilon_2 = \mathsf{d}^{\mathsf{a} \times \mathsf{b}} \times \mathsf{a} \times \mathsf{c}^{\mathsf{b} \times \mathsf{a}}$, which satisfy the requirement that the base of all exponents is a primitive variable. In computing $g(\epsilon_1)$, function α will assign unique primes to the primitive variables a, $p_1 = \alpha(\mathsf{a})$, and b, $p_2 = \alpha(\mathsf{b})$. Then, unique primes are allocated to the sub-expressions involving exponentiation $p_3 = \alpha(\mathsf{c}, p_1 p_2)$ and $p_4 = \alpha(\mathsf{d}, p_1 p_2)$. Thus, $g(\epsilon_1) = p_1 p_3 p_4$. By applying (3.1), we see that

$$\begin{aligned} g(\mathsf{c}^{\mathsf{a} \times \mathsf{b}}) &= \alpha(\mathsf{c}, g(\mathsf{a} \times \mathsf{b})) = \alpha(\mathsf{c}, p_1 p_2) = \alpha(\mathsf{c}, p_2 p_1) \\ &= \alpha(\mathsf{c}, g(\mathsf{b} \times \mathsf{a})) = g(\mathsf{c}^{\mathsf{b} \times \mathsf{a}}). \end{aligned}$$

Similarly, $g(\mathsf{d}^{\mathsf{b} \times \mathsf{a}}) = g(\mathsf{d}^{\mathsf{a} \times \mathsf{b}})$. Therefore, $g(\epsilon_1) = g(\epsilon_2)$.

It is rather straightforward to apply the axioms so that the base of all exponents is a primitive variable. In each step, we apply one of the following transformations to simplify the base:

$$\begin{aligned} \mathbf{1}^e &\Rightarrow \mathbf{1}, \\ (e_1^{e_2})^{e_3} &\Rightarrow e_1^{e_2 \times e_3}, \\ (e_1 \times e_2)^{e_3} &\Rightarrow e_1^{e_3} \times e_2^{e_3}. \end{aligned}$$

The last transformation must be applied with care, since it makes a duplicate occurrence of subexpression e_3.

We shall therefore define an auxiliary function $f : \mathcal{E} \times \mathbb{N} \mapsto \mathbb{N}$ to force the correct evaluation order on g. Function f will make sure that the encoding of the exponent is computed before the encoding of the base. Specifically, we have that $g(e^{e'}) = f(e, g(e'))$.

In evaluating the above, we first compute $\eta = g(e')$. Then, to evaluate $f(e, \eta)$, we need to examine e in greater detail. In doing so, we bear in mind that e is part of an encompassing expression, $e^{e'}$, and that η was previously determined to be the encoding of the exponent e'. There are four cases to consider:

1. The simplest case is that $e = \mathbf{1}$. In this case, $f(\mathbf{1}, \eta) = g(\mathbf{1}^{e'}) = g(\mathbf{1}) = 1$.
2. Another case in which there are no recursive calls is that $e = x$, $x \in \mathcal{X}$. In this case, $f(x, \eta) = g(x^{e'})$. By the definition of the assignment function (3.1) we have

$$f(x, \eta) = \alpha(x, g(e')) = \alpha(x, \eta).$$

3. If $e = e_1 \times e_2$, then

$$\begin{aligned} f(e_1 \times e_2, \eta) &= g\left((e_1 \times e_2)^{e'}\right) = g\left(e_1^{e'} \times e_2^{e'}\right) \\ &= g\left(e_1^{e'}\right) \cdot g\left(e_2^{e'}\right) = f(e_1, \eta) \cdot f(e_2, \eta). \end{aligned}$$

4. Similarly, if $e = e_1^{e_2}$, then

$$
\begin{aligned}
f(e_1^{e_2}, \eta) &= g\left((e_1^{e_2})^{e'}\right) = g\left(e_1^{e_2 \times e'}\right) = f(e_1, g(e_2 \times e')) \\
&= f(e_1, g(e_2) \cdot g(e')) = f(e_1, g(e_2) \cdot \eta).
\end{aligned}
$$

For the formal definition of the encoding, observe that $g(e) = g(e^{\mathbf{1}}) = f(e, g(\mathbf{1})) = f(e, 1)$ and that function f is completely specified by the four above cases.

Definition 3.1. The Gödel encoding of an expression e is defined by $g(e) = f(e, 1)$ where

$$
f(e, \eta) = \begin{cases} 1 & \text{if } e = \mathbf{1} \\ \alpha(x, \eta) & \text{if } e = x \\ f(e_1, \eta) \cdot f(e_2, \eta) & \text{if } e = e_1 \times e_2 \\ f(e_1, \eta \cdot g(e_2)) & \text{if } e = e_1^{e_2} \end{cases} \tag{3.2}
$$

and $\alpha(x, \eta)$ is a function returning a unique prime for each x and η.

Theorem 3.2. If $\epsilon_1 = \epsilon_2$, then $g(\epsilon_1) = g(\epsilon_2)$.

PROOF. By induction on the derivation sequence. We will show that $f(\epsilon_1, \eta) = f(\epsilon_2, \eta)$ for all $\eta \in \mathbb{N}$, from which it follows that $f(\epsilon_1, 1) = f(\epsilon_2, 1)$, and hence that $g(\epsilon_1) = g(\epsilon_2)$.

The proof is by induction on the length of the derivation sequence ending with $\epsilon_1 = \epsilon_2$. The first step in such a derivation must be an instance of one of the axioms. Suppose (for example) that $\epsilon_1 = \epsilon_2$ is an instance of the Distributivity axiom. Then, there must be expressions A, B and C such that ϵ_1 is in the form $(A \times B)^C$, while ϵ_2 is in the form $(A^C) \times (B^C)$. It is straightforward to apply the recursive definition of f (Definition 3.1) to $f(\epsilon_1, \eta)$

$$
\begin{aligned}
f\big((A \times B)^C, \eta\big) &= f(A \times B, \eta \cdot g(C)) \\
&= f(A, \eta \cdot g(C)) \cdot f(B, \eta \cdot g(C)),
\end{aligned}
$$

and $f(\epsilon_2, \eta)$

$$
\begin{aligned}
f\big(A^C \times B^C, \eta\big) &= f(A^C, \eta) \cdot f(B^C, \eta) \\
&= f(A, \eta \cdot g(C)) \cdot f(B, \eta \cdot g(C)),
\end{aligned}
$$

and we see that they are equal. The cases corresponding to the other axioms $\mathcal{A}.0$–$\mathcal{A}.6$ are similar.

The induction step is that $\epsilon_1 = \epsilon_2$ is either an axiom (a case we covered in the induction base) or the result of applying one of the four inference rules on previous equalities. The cases of the Symmetry and Transitivity inference rules immediately follow from equality of natural numbers. Suppose now that $\epsilon_1 = \epsilon_2$ resulted from applying Congruence of exponentiation, i.e., $\epsilon_1 = A^C$, $\epsilon_2 = B^D$, $A = B$ and $C = D$. From the induction hypothesis we have that for all $\eta' \in \mathbb{N}$, $f(A, \eta') = f(B, \eta')$ and $f(C, \eta') = f(D, \eta')$. Specifically, for $\eta' = 1$, we have that $g(C) = g(D)$. Therefore,

$$
\begin{aligned}
f(\epsilon_1, \eta) &= f(A^C, \eta) = f(A, \eta \cdot g(C)) = f(A, \eta \cdot g(D)) \\
&= f(B, \eta \cdot g(D)) = f(B^D, \eta) = f(\epsilon_2, \eta).
\end{aligned}
$$

The case of Congruence of $\times$ is similar. $\square$

We now proceed to define g^{-1}, the inverse encoding, in such a way that $\forall e \in \mathcal{E}, g^{-1} \circ g(e) = e$. (Note that the equality in the above does not mean that $g^{-1} \circ g(e)$ has the same structure as e, but rather that it can be brought to the form e by a derivation sequence.) To this end, we first need to extend the assignment function α to be a bijection. In words, for each prime p, $\alpha^{-1}(p)$ returns a pair $\langle x, \eta\rangle$, such that $\alpha(x, \eta) = p$. This extension is possible since the cardinality of $\mathbb{P}$ is $\aleph_0$, which is also the cardinality of $\mathcal{X} \times \mathbb{N}$.

Definition 3.3. The partial function $g^{-1}(\eta)$ is defined as follows. If $\eta = 1$, then $g^{-1}(\eta) = \mathbf{1}$. Otherwise, η has a unique prime factorization $\eta = p_1 \cdots p_k$, $p_i \le p_{i+1}$ for $i = 1, \dots, k-1$. Let $\alpha^{-1}(p_i) = \langle x_i, \eta_i\rangle$, for $i = 1, \dots, k$. Then,

$$g^{-1}(\eta) = x_1^{g^{-1}(\eta_1)} \times \cdots \times x_k^{g^{-1}(\eta_k)}.$$

Lemma 3.4. For all $e, e' \in \mathcal{E}$, $g^{-1} \circ g(e^{e'})$ exists. Moreover, $e^{e'} = g^{-1} \circ g(e^{e'})$.

Proof. By Definition 3.1, $g(e^{e'}) = f(e^{e'}, 1) = f(e, 1 \cdot g(e')) = f(e, g(e'))$. It is therefore sufficient to prove that $g^{-1} \circ f(e, g(e'))$ exists and that

$$e^{e'} = g^{-1} \circ f(e, g(e')).$$

We will prove this claim by induction on $|e^{e'}|$.

To evaluate $f(e, g(e'))$, we must examine the structure of e according to Definition 3.1. There are four cases to consider.

1. The inductive base is the case $e = \mathbf{1}$. In this case,

$$\mathbf{1}^{e'} = \mathbf{1} = g^{-1}(1) = g^{-1} \circ f(\mathbf{1}, g(e')).$$

2. If $e = x$, i.e., the input takes the form $x^{e'}$, we shall first use the inductive hypothesis on the expression $(e')^{\mathbf{1}}$ to show that $e' = g^{-1} \circ g(e')$. (This is safe to do since by Definition 2.1 $|\mathbf{1}| = 1$, $|x| = 2$ and hence $|(e')^{\mathbf{1}}| < |x^{e'}|$.) We have

$$e' = (e')^{\mathbf{1}} = g^{-1} \circ f(e', g(\mathbf{1})) = g^{-1} \circ f(e', 1) = g^{-1} \circ g(e').$$

Also, by definition $f(x, g(e')) = \alpha(x, g(e'))$. Since number $\alpha(x, g(e'))$ is prime, by Definition 3.3

$$x^{e'} = x^{g^{-1} \circ g(e')} = g^{-1} \circ \alpha(x, g(e')) = g^{-1} \circ f(x, g(e')).$$

3. In the case that $e = e_1 \times e_2$, we have that $|e_1|, |e_2| < |e|$. Therefore, we can apply the inductive hypothesis on $e_i^{e'}$, $i = 1, 2$, to obtain that $g^{-1} \circ f(e_i, g(e'))$ exists and that

$$e_i^{e'} = g^{-1} \circ f(e_i, g(e')). \tag{3.3}$$

Also, Definition 3.3 implies that if both $g^{-1}(\eta_1)$ and $g^{-1}(\eta_2)$ exist, then $g^{-1}(\eta_1 \cdot \eta_2)$ exists and

$$g^{-1}(\eta_1) \times g^{-1}(\eta_2) = g^{-1}(\eta_1 \cdot \eta_2). \tag{3.4}$$

Applying (3.3) and then (3.4) we obtain

$$\begin{aligned}(e_1 \times e_2)^{e'} &= e_1^{e'} \times e_2^{e'} \\ &= \left[g^{-1} \circ f(e_1, g(e'))\right] \times \left[g^{-1} \circ f(e_2, g(e'))\right] \\ &= g^{-1}\left[f(e_1, g(e')) \cdot f(e_2, g(e'))\right],\end{aligned}$$

and by Definition 3.1

$$= g^{-1} \circ f(e_1 \times e_2, g(e')).$$

4. The remaining case is $e = e_1^{e_2}$. By the Currying axiom we can rewrite the input as $(e_1^{e_2})^{e'} = e_1^{e_2 \times e'}$. Although $e_1^{e_2 \times e'}$ has the same number of nodes as $(e_1^{e_2})^{e'}$, it has one fewer exponentiation operator. Therefore, by Definition 2.1, $|e_1^{e_2 \times e'}| < |(e_1^{e_2})^{e'}|$, and by the inductive hypothesis

$$e_1^{e_2 \times e'} = g^{-1} \circ f(e_1, g(e_2 \times e')). \tag{3.5}$$

From Definition 3.1 we have that

$$\begin{aligned}g(e_2 \times e') = f(e_2 \times e', 1) &= f(e_2, 1) \cdot f(e', 1) \\ &= g(e_2) \cdot g(e').\end{aligned} \tag{3.6}$$

Substituting (3.6) into (3.5)

$$e_1^{e_2 \times e'} = g^{-1} \circ f(e_1, g(e_2) \cdot g(e')).$$

Commuting the operands of $\cdot$ and applying Definition 3.1

$$e_1^{e_2 \times e'} = g^{-1} \circ f(e_1^{e_2}, g(e')).$$

The proof ends by recalling that $(e_1^{e_2})^{e'} = e_1^{e_2 \times e'}$. □

Theorem 3.5. If $g(\epsilon_1) = g(\epsilon_2)$, then $\epsilon_1 = \epsilon_2$.

PROOF. From Lemma 3.4, we have that, for $i = 1, 2$

$$\begin{aligned}\epsilon_i = \epsilon_i^{\mathbf{1}} &= g^{-1} \circ g\big(\epsilon_i^{\mathbf{1}}\big) = g^{-1} \circ f\big(\epsilon_i^{\mathbf{1}}, 1\big) \\ &= g^{-1} \circ f(\epsilon_i, g(\mathbf{1})) = g^{-1} \circ f(\epsilon_i, 1) = g^{-1} \circ g(\epsilon_i).\end{aligned}$$

Since $g(\epsilon_1) = g(\epsilon_2)$, we have that $\epsilon_1 = g^{-1} \circ g(\epsilon_1) = g^{-1} \circ g(\epsilon_2) = \epsilon_2$. □

4. Monte Carlo algorithm

Checking two expressions $\epsilon_1, \epsilon_2 \in \mathcal{E}$, we know that they are equal precisely when $g(\epsilon_1) = g(\epsilon_2)$. Our concern lies with an efficient computation of the function g with respect to the problem size $n = |\epsilon_1| + |\epsilon_2|$.

Observe first that the assignment function α is invoked at most n times, and that it can be easily implemented to return primes from the sequence of the first n primes $p_1 = 2, p_2 = 3, p_3 = 5, \ldots, p_n$ in order. This function makes use of a hash-table data structure H. The implementation checks whether a prime is already associated with the key $\langle x, \eta\rangle$ in H, in which case this prime is returned. Otherwise, α picks the next unselected prime, associates it with the key, and returns this prime.

Since lookups and insertions in H can be carried out in constant time [Dietzfelbinger et al., 1994] we have that $\alpha(x, \eta)$ requires constant time. It is also easy to see that the recursive Definition 3.1 is such that there are at most n steps in computing $g(\epsilon_1)$, $g(\epsilon_2)$. Each of these steps may involve at most two recursive calls, and a single integer multiplication. The bottleneck in our computation is therefore integer multiplication. If we represent intermediate values as $(p_1)^{i_1} \cdots (p_n)^{i_n}$, where $0 \leq i_j \leq n$, $j = 1, \ldots, n$, then each multiplication requires $O(n)$ time, and the entire algorithms runs in quadratic time.

In this section we explore a hashed representation of these values, in which multiplication requires constant time. Specifically, let q be some appropriately chosen prime, and let g_q, f_q be the variants of functions g and f (Definition 3.1) obtained by using multiplication in $\mathbb{Z}_q$ (i.e., modulo q). If $q \in n^{O(1)}$ then g_q can be computed in linear time.

It is easy to see that $g(e_1) = g(e_2) \Rightarrow g_q(e_1) = g_q(e_2)$. However, the reverse is not always true because the modulo operation might cause a *collision*. Intuitively, we will show that for every two expressions e_1, e_2, $n = |e_1| + |e_2|$, the number of "bad" primes q that causes a collision is less than n^3. Thus if we randomly choose a prime smaller than n^5 then the probability that we chose a bad prime is less than $\frac{1}{n}$.

Algorithm 1 shows how an expression can be recursively traversed to compute the hashed Gödel encoding. Function V_q on the right hand side will be used in the following section.

Comparing function $F_q(e, r = 0)$ with Definition 3.1, we can easily verify that it computes $f_q(e, \boldsymbol{\eta}_q[r])$. Moreover, $F_\infty(e, r) = f(e, \boldsymbol{\eta}_\infty[r])$. We use a default notation for arguments, so $F_q(e) = F_q(e, 0) = f_q(e, \boldsymbol{\eta}_q[0]) = f_q(e, 1) = g_q(e)$. Clearly, $F_\infty(e) = g(e)$.

As can easily be seen, the number of recursive calls is linear. The global index v is incremented only in line 9, and a new entry is added to array $\boldsymbol{\eta}_q$ only on line 10. Therefore, at most $n + 1$ entries are computed and stored in array $\boldsymbol{\eta}_q$. With the standard assumption that arithmetical and comparison operations on numbers with logarithmic number of bits can be carried out in constant time we have,

Lemma 4.1. If $|e| = n$ and $q \in n^{O(1)}$, then the invocation $F_q(e)$ runs in $O(n)$ time and consumes $O(n)$ space.

Algorithm 2 presents procedure $M_d(\epsilon_1, \epsilon_2)$ for determining whether $\epsilon_1 = \epsilon_2$.

In the preprocessing stage (line 2), the algorithm computes the sequence of the first n primes using, e.g., Eratosthenes sieve in

$$O(n \log n / \log \log n)$$

time [Pritchard, 1981]. Line 6 then chooses a prime q uniformly and at random from the primes in the range $(p_n, n^{d+3}]$.

This random selection can be implemented in a number of ways. For example, one can compute all primes in this range in a polynomial time charged to the preprocessing stage. The algorithm may also repeatedly choose a random number in the range $(p_n, n^{d+3}]$ until a prime number is thus selected. It follows from the prime numbers theorem that this process yields a prime in a logarithmic number of expected iterations, while the nature of the process guarantees that the selection is uniform. (Checking whether a number is prime can be done with standard poly-

Algorithm 1 Recursive computation of hashed forms of the Gödel encoding. The recursive functions below use a global array of integers $\boldsymbol{\eta}_q[0 \ldots n]$. Also, a global variable v is maintained for counting the recursive steps. Initially, $v = 0$, and $\boldsymbol{\eta}_q[0] = 1$. Both functions take as parameters an expression e and an index $r \in [0, n]$, and compute the hashed encoding of $e^{e'}$, where the encoding of e' was computed in the r^{th} recursive computation step. The principal function is $F_q(e, r = 0)$, which is used for computing g_q. Function $V_q(e, r = 0)$ whose structure is similar to that of $F_q(e, r = 0)$ may be invoked after the entire recursive run of F_q. It makes use of two additional global arrays $\boldsymbol{\varphi}_q[1 \ldots n]$ of multi-sets of primes, and $\boldsymbol{\rho}_q[1 \ldots n]$ of indices of previous computation steps. This function will be discussed in detail in the next section.

```
    fun $F_q(e, r = 0)$
    Returns $f_q(e, \boldsymbol{\eta}_q[r])$

 1: If $e = \mathbf{1}$ then
 2:    return $1$
 3: else if $e = x$ then
 4:    return $\alpha(x, \boldsymbol{\eta}_q[r])$
 5: else if $e = e_1 \times e_2$ then
 6:    return $F_q(e_1, r) \cdot F_q(e_2, r) \bmod q$
 7: else if $e = e_2^{e_1}$ then
 8:    let $\mu \leftarrow F_q(e_1)$
 9:    $v \leftarrow v + 1$
10:    $\boldsymbol{\eta}_q[v] \leftarrow \boldsymbol{\eta}_q[r] \cdot \mu \bmod q$
11:    return $F_q(e_2, v)$
```

```
    fun $V_q(e, r = 0)$
    Returns a multi-set $\omega$,
    s.t. $F_q(e, r) = \left(\prod_{p \in \omega} p\right) \bmod q$
 1: If $e = \mathbf{1}$ then
 2:    return $\emptyset$
 3: else if $e = x$ then
 4:    return $\{\alpha(x, \boldsymbol{\eta}_q[r])\}$
 5: else if $e = e_1 \times e_2$ then
 6:    return $V_q(e_1, r) \cup V_q(e_2, r)$
 7: else if $e = e_2^{e_1}$ then
 8:    let $\psi \leftarrow V_q(e_1)$
 9:    $v \leftarrow v + 1$
10:    $\boldsymbol{\rho}_q[v] \leftarrow r$;    $\boldsymbol{\varphi}_q[v] \leftarrow \psi$
11:    return $V_q(e_2, v)$
```

Algorithm 2 $M_d(\epsilon_1, \epsilon_2)$ a Monte-Carlo algorithm for deciding whether $\epsilon_1 = \epsilon_2$

```
 1: global $\langle p_1, \ldots, p_n \rangle$, $v$, $\boldsymbol{\eta}_q[0 \ldots n]$
 2: (Preprocessing) $\langle p_1, \ldots, p_n \rangle \leftarrow$ the sequence of the first $n$ primes; $\boldsymbol{\eta}_q[0] \leftarrow 1$
 3: If $d = \infty$ then
 4:    let $q \leftarrow \infty$
 5: else
 6:    let $q \leftarrow$ a random prime uniformly chosen from the primes in $(p_n, n^{d+3}]$
 7: $v \leftarrow 0$;    let $\eta_1 \leftarrow F_q(x^{\epsilon_1})$;    let $\eta_2 \leftarrow F_q(x^{\epsilon_2})$
 8: If $\eta_1 \neq \eta_2$ then
 9:    print "$\epsilon_1$ is provably not equal to $\epsilon_2$"
10: else
11:    print "$\epsilon_1$ is (probably) equal to $\epsilon_2$, the error probability is $O(n^{-d} \log n)$"
```

logarithmic algorithms for finding pseudo-primes, or even the new results for primality testing in poly-logarithmic time [Agrawal et al., 2002].)

The algorithm then computes $F_q(x^{\epsilon_1})$ and $F_q(x^{\epsilon_2})$ (line 7) using the sequence $p_1, \ldots, p_n$ for the hash-table implementation of the assignment function α. It then determines (line 8) whether the inputs are equal by comparing the values the two invocations returned.

It is easy to see that $F_q(x^{\epsilon_1}) = F_q(x^{\epsilon_2}) \Leftrightarrow F_q(\epsilon_1) = F_q(\epsilon_2)$. We make the calls $F_q(x^{\epsilon_1})$ and $F_q(x^{\epsilon_2})$ instead of $F_q(\epsilon_1)$ and $F_q(\epsilon_2)$ to make sure that the values returned by $F_q(\epsilon_1)$ and $F_q(\epsilon_2)$ are stored in the array $\boldsymbol{\eta}_q$.

The *canonical* run of M_d and F_q is a hypothetical execution with $d = q = \infty$. This run may generate very large numbers, and require quadratic time. However, as it is easy to see, the invocation $M_\infty(\epsilon_1, \epsilon_2)$ computes $g(x^{\epsilon_1})$ and $g(x^{\epsilon_2})$ *precisely* and therefore cannot err in determining whether the two inputs are equal. We say that the hypothetical canonical run leaves a *trace* $\boldsymbol{\eta}_\infty$ while invoking F_∞. For brevity, we omit the ∞ symbol when it occurs in subscripts and write M, F and $\boldsymbol{\eta}$ instead of M_∞, F_∞ and $\boldsymbol{\eta}_\infty$.

Our main concern lies with the case of a fixed $d > 0$, which we will analyze by comparing it to the canonical run.

Theorem 4.2. For all fixed $d > 0$, $M_d(\epsilon_1, \epsilon_2)$ determines (after a pre-processing stage, depending solely on n) whether $\epsilon_1 = \epsilon_2$ in time $O(n)$. The algorithm may only err in determining that the two input expressions are equal when they are not, an event which may occur with probability $O(n^{-d} \log n)$.

The remainder of this section is dedicated to the proof. The time complexity follows from Lemma 4.1. For correctness, we first argue that if Algorithm 2 reaches line 9 then the input expressions cannot be equal.

Lemma 4.3. For all $q \geq 2$, if $\epsilon_1 = \epsilon_2$ then $F_q(\epsilon_1) = F_q(\epsilon_2)$.

PROOF. The proof is identical to the proof of Theorem 3.2, except that all multiplications are carried out $\bmod q$. All we rely on is the commutative and associative nature of multiplication. $\square$

Lemma 4.3 is not useful on its own, since the hashed encoding may be the same for many unequal expressions. An extreme case in point is $q = 2$, in which g_q partitions $\mathcal{E}$ into two equivalence classes. Henceforth we assume that $\epsilon_1 \neq \epsilon_2$, (and hence $g(\epsilon_1) \neq g(\epsilon_2)$), and try to characterize those q for which $F_q(\epsilon_1) \neq F_q(\epsilon_2)$.

The run of $M_q(\epsilon_1, \epsilon_2)$ left a trace in the array $\boldsymbol{\eta}_q$, and moreover

$$F_q(\epsilon_1) = \boldsymbol{\eta}_q[v_1],$$
$$F_q(\epsilon_2) = \boldsymbol{\eta}_q[v_2],$$

for some v_1 and v_2. We can bound the error probability by comparing this trace $\boldsymbol{\eta}_q$ with the trace of the canonical run $\boldsymbol{\eta}$.

Definition 4.4. Prime q is *collision-free* (with respect to ϵ_1 and ϵ_2) if for all $\boldsymbol{\eta}[r_1] \neq \boldsymbol{\eta}[r_2]$ it also holds that $(\boldsymbol{\eta}[r_1] \neq \boldsymbol{\eta}[r_2]) \bmod q$. Otherwise q is *colliding*.

Lemma 4.5. Let $q > p_n$ be a collision-free prime. Then, for all $0 \leq r \leq n$, $\boldsymbol{\eta}_q[r] = \boldsymbol{\eta}[r] \bmod q$.

For the proof we need the notion of a *stack log* of an algorithmic statement, which is the log of all *invocation* and *return* commands which execute when this statement is executed. Invocation commands include the name of the invoked routine (function or procedure), and its arguments. A return command does *not* include the return value.

PROOF. Observe first that the stack log of a statement $v \leftarrow 0; F_q(e)$ is the same as that of a statement $v \leftarrow 0; F(e)$, except that calls to F_q are replaced by calls to F. We show by induction that in this log $F_q(e, r) = F(e, r) \bmod q$.

As usual, there are four cases to consider

1. If $e = \mathbf{1}$, then obviously both functions return 1.
2. If $e = x$, then F_q returns $\alpha(x, \boldsymbol{\eta}_q[r])$ while F returns $\alpha(x, \boldsymbol{\eta}[r])$. From the inductive hypothesis we have that $\boldsymbol{\eta}_q[r] = \boldsymbol{\eta}[r] \bmod q$. Moreover, since q is collision free, it follows that H (the hash table of F) is isomorphic to H_q (the hash table of F_q) in the sense that key $\langle x', \boldsymbol{\eta}[r'] \rangle$ occurs in H and is mapped by it to prime p, if and only if $\langle x', \boldsymbol{\eta}[r'] \bmod q \rangle$ occurs in H_q and is mapped by it to p. Thus, if $\langle x, \boldsymbol{\eta}[r] \rangle \in \mathrm{H}$, then obviously $\alpha(x, \boldsymbol{\eta}[r]) = \alpha(x, \boldsymbol{\eta}[r] \bmod q)$. If on the other hand $\langle x, \boldsymbol{\eta}[r] \rangle \notin \mathrm{H}$, we rely on $\alpha(x, \boldsymbol{\eta}[r])$ deterministically returning the next prime and on $q \geq p_n$ to make the inductive step.
3. If $e = e_1 \times e_2$, then from the inductive hypothesis we have that

$$\begin{aligned} F_q(e_1, r) &= F(e_1, r) \bmod q, \\ F_q(e_2, r) &= F(e_2, r) \bmod q. \end{aligned}$$

   Thus,

$$\begin{aligned} F_q(e_1 \times e_2, r) &= F_q(e_1, r) \cdot F_q(e_2, r) \bmod q \\ &= (F(e_1, r) \bmod q) \cdot (F(e_2, r) \bmod q) \bmod q \\ &= F(e_1, r) \cdot F(e_2, r) \bmod q \\ &= F(e_1 \times e_2, r) \bmod q. \end{aligned}$$

4. If $e = e_2^{e_1}$, then from the inductive hypothesis we have that $F_q(e_1) = F(e_1) \bmod q$, and $\boldsymbol{\eta}_q[r] = \boldsymbol{\eta}[r] \bmod q$. Now F_q and F compute and store $\boldsymbol{\eta}_q[v]$ and $\boldsymbol{\eta}[v]$,

$$\begin{aligned} \boldsymbol{\eta}_q[v] &= \boldsymbol{\eta}_q[r] \cdot F_q(e_1) \bmod q = \boldsymbol{\eta}[r] \cdot F(e_1) \bmod q \\ &= \boldsymbol{\eta}[v] \bmod q. \end{aligned}$$

   Thus,

$$\begin{aligned} F_q(e_2^{e_1}, r) &= F_q(e_2, v) = F(e_2, v) \bmod q \\ &= F(e_2^{e_1}, v) \bmod q. \quad \square \end{aligned}$$

Corollary 4.6. Let $q > p_n$ be a collision-free prime. Then, $\boldsymbol{\eta}[r_1] \neq \boldsymbol{\eta}[r_2] \Rightarrow \boldsymbol{\eta}_q[r_1] \neq \boldsymbol{\eta}_q[r_2]$.

Lemma 4.7. Let $q > p_n$ be a collision-free prime. Then $\epsilon_1 \neq \epsilon_2 \Rightarrow F_q(\epsilon_1) \neq F_q(\epsilon_2)$.

PROOF. Recall that $\boldsymbol{\eta}[v_1] = F(\epsilon_1) \neq F(\epsilon_2) = \boldsymbol{\eta}[v_2]$. From Corollary 4.6, we have that $\boldsymbol{\eta}_q[v_1] \neq \boldsymbol{\eta}_q[v_2]$. The lemma follows from noting that $F_q(\epsilon_1) = \boldsymbol{\eta}_q[v_1]$ and $F_q(\epsilon_2) = \boldsymbol{\eta}_q[v_2]$. $\square$

Let $Q \subset \mathbb{P}$ be the set of *colliding* primes greater than p_n.

Lemma 4.8. Let q be a prime number chosen uniformly and at random from all primes in the range $(p_n, n^{d+3}]$. Then, the probability that $q \in Q$ is $O(n^{-d} \log n)$.

PROOF. For each colliding prime $q \in Q$, there exists a pair $\boldsymbol{\eta}[r_1]$ and $\boldsymbol{\eta}[r_2]$, such that $\boldsymbol{\eta}[r_1] > \boldsymbol{\eta}[r_2]$ but $(\boldsymbol{\eta}[r_1] = \boldsymbol{\eta}[r_2]) \bmod q$, i.e., q is a divisor of $\boldsymbol{\eta}[r_1] - \boldsymbol{\eta}[r_2] > 0$. Since the evaluation of F involves at most n multiplications, $\boldsymbol{\eta}[r_1], \boldsymbol{\eta}[r_2] \leq (p_n)^n$. Hence, $\boldsymbol{\eta}[r_1] - \boldsymbol{\eta}[r_2] \leq (p_n)^n$ and it can have at most n prime divisors greater than p_n. Noting that there are at most n^2 such pairs we obtain that $|Q| \leq n^3$.

The lemma follows by noting that from the prime numbers theorem, the number of primes in the range $(p_n, n^{d+3}]$ is $O(n^{d+3}/\log n)$. $\square$

5. Las Vegas algorithm

In the case of an unfortunate selection of q (i.e., $q \in Q$), if Algorithm 2 finds that $F_q(x^{\epsilon_1}) = F_q(x^{\epsilon_2})$ then it may erroneously conclude that $\epsilon_1 = \epsilon_2$. This section describes an algorithm which uses the trace $\boldsymbol{\eta}_q$ of the two invocations $F_q(x^{\epsilon_1})$ and $F_q(x^{\epsilon_2})$, to determine in $O(n \log n)$ time whether $q \in Q$.

Each number in the trace $\boldsymbol{\eta}_q[1 \ldots n]$ is computed from a multiplication modulo q. In the Las Vegas algorithm we need to *prove* that no collisions occurred. (If a collision did occur then we simply randomly pick another prime, and repeat until we find a collision-free prime.) To prove that no collisions occurred we invoke another function V_q that re-iterates the steps of the function F_q and stores multi-sets of primes into an array $\boldsymbol{\phi}_q(1 \ldots n)$. The multiplication of elements in the multi-set $\boldsymbol{\phi}_q(i)$ is equal modulo q to $\boldsymbol{\eta}_q[i]$, thus $\boldsymbol{\phi}_q(i) = \boldsymbol{\phi}_q(j) \Rightarrow \boldsymbol{\eta}_q[i] = \boldsymbol{\eta}_q[j]$. To verify that q is indeed collision-free we need to check that we had no collisions due to the modulo operation, i.e., $\boldsymbol{\phi}_q(i) \neq \boldsymbol{\phi}_q(j) \Rightarrow \boldsymbol{\eta}_q[i] \neq \boldsymbol{\eta}_q[j]$.

The catch is that the sum of multi-sets sizes is $O(n^2)$. Luckily the multi-sets have many elements in common and we can represent them in linear space. Specifically, instead of storing directly the array $\boldsymbol{\phi}_q(i)$ we represent it using a tree structure: $\boldsymbol{\varphi}_q[1 \ldots n]$ stores a multi-set for each tree node, and $\boldsymbol{\rho}_q[1 \ldots n]$ stores the index of the parent of the node. More precisely,

Definition 5.1. For a node u, the *expanded* multi-set of primes associated with this node, $\boldsymbol{\phi}_q(u)$, is defined recursively as follows:

$$\boldsymbol{\phi}_q(u) = \begin{cases} \emptyset & \text{if } u = 0 \\ \boldsymbol{\varphi}_q[u] \cup \boldsymbol{\phi}_q(\boldsymbol{\rho}_q[u]) & \text{otherwise.} \end{cases}$$

Also, let $\pi_q(u)$ be the product of all members of $\boldsymbol{\phi}_q(u)$, i.e., $\pi_q(u) = \prod_{p \in \boldsymbol{\phi}_q(u)} p$.

We will prove that (i) this compact representation as a tree requires only linear space (Lemma 5.2), (ii) $\pi_q(u) \bmod q = \boldsymbol{\eta}_q[u]$ (Lemma 5.4), and (iii) prime q is colliding if two nodes i, j exist such that $\boldsymbol{\phi}_q(i) \neq \boldsymbol{\phi}_q(j)$ but $\boldsymbol{\eta}_q[i] = \boldsymbol{\eta}_q[j]$ (Lemma 5.5). Determining if q is colliding can be done in $O(n \log n)$ time while the space is $O(n)$ [Zibin et al., 2003, Section 5].

Algorithm 3 presents our randomized Las-Vegas algorithm for determining whether $\epsilon_1 = \epsilon_2$. The algorithm starts with the same preprocessing stage as in Algorithm 2. Also, in the main loop (which is expected to iterate $1 + o(1)$ times) the algorithm makes a similar invocation of $F_q(x^{\epsilon_1})$ and $F_q(x^{\epsilon_2})$.

As before, if the values these invocations return are distinct, then the two input expressions are provably not equal (lines 6–8). Otherwise, i.e., in the case that $F_q(x^{\epsilon_1}) = F_q(x^{\epsilon_2})$, the algorithm must verify that q is indeed collision free, i.e., $q \notin Q$. To do so, the algorithm invokes function V_q twice at line 9 to reiterate the steps of the invocations of F_q in line 5, collecting a more detailed trace of the computation. This information is then used by the loop condition (line 10) to check whether $q \in Q$.

If the algorithm discovers at this point that $q \in Q$, then another iteration must be made in which a new random q is selected. Otherwise, it concludes that q is collision free, and hence the inputs must be equal (line 11).

Algorithm 3 $L(\epsilon_1, \epsilon_2)$ a Las-Vegas algorithm for deciding whether $\epsilon_1 = \epsilon_2$

```
1: global $\langle p_1, \ldots, p_n \rangle$, $v$, $\boldsymbol{\eta}_q[0 \ldots n]$, $\boldsymbol{\varphi}_q[1 \ldots n]$, $\boldsymbol{\rho}_q[1 \ldots n]$
2: (Preprocessing) $\langle p_1, \ldots, p_n \rangle \leftarrow$ the sequence of the first $n$ primes; $\boldsymbol{\eta}_q[0] \leftarrow 1$
3: repeat
4:     let $q \leftarrow$ a random prime uniformly chosen from the primes in $(p_n, n^4]$
5:     $v \leftarrow 0$;   let $\eta_1 \leftarrow F_q(x^{\epsilon_1})$;   let $\eta_2 \leftarrow F_q(x^{\epsilon_2})$
6:     If $\eta_1 \neq \eta_2$ then
7:         print "$\epsilon_1$ is provably not equal to $\epsilon_2$"
8:         return
9:     $v \leftarrow 0$;   call $V_q(x^{\epsilon_1})$;   call $V_q(x^{\epsilon_2})$
10: until For all $i, j \in [1, n]$, $\boldsymbol{\phi}_q(i) \neq \boldsymbol{\phi}_q(j) \Rightarrow \boldsymbol{\eta}_q[i] \neq \boldsymbol{\eta}_q[j]$ // loop until we have
    no collisions, i.e., $q \notin Q$
11: print "$\epsilon_1$ is provably equal to $\epsilon_2$"
```

We now describe in greater detail function V_q, then the information it gathers, and finally how this information can be used to test whether $q \in Q$.

Algorithm 1 depicts functions V_q and F_q side-by-side, showing that they share the same control flow structure and the same recursive call patterns. We also see that both functions use the global variable v in exactly the same fashion. Observe that the calls in Algorithm 3 to F_q (line 5) and to V_q (line 9) have identical parameters, and start with the same initial value of v. Therefore, the stack log of line 5 is the same as the stack log of line 9 in substituting F_q for V_q.

Function V_q manipulates multi-set of values (specifically, primes). We represent these as bi-directional linked-lists in memory. Therefore, assignment requires constant time. In line 6, the function computes the union of two such multi-sets. The constant time implementation is by a destructive concatenation of the lists returned by the two previous recursive calls. We therefore have,

Lemma 5.2. If $|e| = n$ and $q \in n^{O(1)}$, then the invocation $V_q(e)$ runs in $O(n)$ time and consumes $O(n)$ space.

We argue that the value returned by F_q is the product modulo q of the multi-set of values returned by V_q. More precisely,

Lemma 5.3. Suppose that F_q returned an integer μ in step i of the stack log of line 5, and that V_q returned a multi-set ψ in step i of the stack log of line 9. Then, $\mu = \left(\prod_{p\in\psi} p\right) \bmod q$.

PROOF. Trivial by induction on i and a side-by-side comparison of the two functions. □

An important difference between the two functions lies in line 10. The trace $\boldsymbol{\eta}_q$ recorded by function F_q is significantly enhanced here by V_q. In fact, the invocation $V_q(e)$ generates what we call the *Gödel tree* which is the standard symbolic expression representation of $g_q(e)$.

To understand better how this tree is related to the input expression, consider the following expression and its expansion.

$$\begin{aligned}&\left(\left(\mathsf{u}_1^{\mathsf{a}_1\mathsf{a}_2\mathsf{a}_2\mathsf{a}_3}\mathsf{u}_2^{\mathsf{b}_1\mathsf{b}_2}\right)^{\mathsf{c}_1\mathsf{c}_2}\mathsf{u}_3^{(\mathsf{xy})^\mathsf{d}}\right)^{\mathsf{e}_1\mathsf{e}_2}\left(\mathsf{u}_4^{\mathsf{f}}\mathsf{u}_5^{\mathsf{g}_1\mathsf{g}_2}\right)^{\mathsf{h}} = \\ &\mathsf{u}_1^{\mathsf{a}_1\mathsf{a}_2\mathsf{a}_2\mathsf{a}_3\mathsf{c}_1\mathsf{c}_2\mathsf{e}_1\mathsf{e}_2}\mathsf{u}_2^{\mathsf{b}_1\mathsf{b}_2\mathsf{c}_1\mathsf{c}_2\mathsf{e}_1\mathsf{e}_2}\mathsf{u}_3^{\mathsf{x}^\mathsf{d}\mathsf{y}^\mathsf{d}\mathsf{e}_1\mathsf{e}_2}\mathsf{u}_4^{\mathsf{fh}}\mathsf{u}_5^{\mathsf{g}_1\mathsf{g}_2\mathsf{h}}\end{aligned} \tag{5.1}$$

The exponents of $\mathsf{u}_1, \ldots, \mathsf{u}_5$ are long products. Figure 5.1 shows that these can be represented more compactly as leaves of a tree. A moment pondering should convince the reader that a similar tree is induced by expanding any expression involving exponentiation and products.

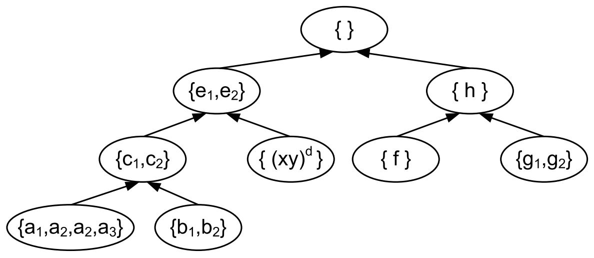

Fig. 5.1. A tree representation of the exponents of $\mathsf{u}_1, \ldots, \mathsf{u}_5$ in the left-hand side of (5.1).

The Gödel tree is similar in principle to the tree of Figure 5.1, except that the members of the multi-set are not primitive variables or compound expressions as in the figure, but rather primes returned by the assignment function. Specifically, a primitive variable x is stored as $\alpha(x)$, while a compound expression is e expanded and brought to the form $e = x_1^{e_1} \times \cdots \times x_k^{e_k}$ (where each e_i is expanded recursively in the same fashion). The multi-set of e stores all $\alpha(x_i, g_q(e_i))$, $i = 1, \ldots, k$. In the example, the node labelled $(\mathsf{xy})^\mathsf{d}$ is stored as $\{\alpha(\mathsf{x}, \alpha(\mathsf{d})), \alpha(\mathsf{y}, \alpha(\mathsf{d}))\}$.

Nodes in the Gödel tree are represented by indices into two global arrays, denoted $\boldsymbol{\rho}_q[1 \ldots n]$ and $\boldsymbol{\varphi}_q[1 \ldots n]$. The tree root is 0, and the parent of any other node v is $\boldsymbol{\rho}_q[v]$. In $\boldsymbol{\varphi}_q[v]$ we store a multi-set of primes associated with v.

Figure 5.2 shows the Gödel tree representation of the expression

$$\left(\left(\mathsf{u}_1^{\mathsf{a}_1\mathsf{a}_2\mathsf{a}_2\mathsf{a}_3}\mathsf{u}_2^{\mathsf{b}_1\mathsf{b}_2}\right)^{\mathsf{c}_1\mathsf{c}_2}\mathsf{u}_3^{(\mathsf{xy})^\mathsf{d}}\right)^{\mathsf{e}_1\mathsf{e}_2}\left(\mathsf{u}_4^{\mathsf{f}}\mathsf{u}_5^{\mathsf{g}_1\mathsf{g}_2}\right)^{\mathsf{h}} \tag{5.2}$$

This tree was obtained by invoking F_q and later V_q on (5.2). The first invocation computed

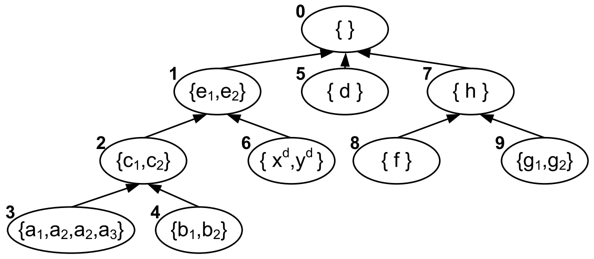

Fig. 5.2. The Gödel tree representation of (5.2).

and stored the global array $\boldsymbol{\eta}_q$:

$$\boldsymbol{\eta}_q[1] = 1 \cdot g_q(\mathsf{e}_1\mathsf{e}_2)$$
$$\boldsymbol{\eta}_q[2] = \boldsymbol{\eta}_q[1] \cdot g_q(\mathsf{c}_1\mathsf{c}_2)$$
$$\boldsymbol{\eta}_q[3] = \boldsymbol{\eta}_q[2] \cdot g_q(\mathsf{a}_1\mathsf{a}_2\mathsf{a}_2\mathsf{a}_3)$$
$$\boldsymbol{\eta}_q[4] = \boldsymbol{\eta}_q[2] \cdot g_q(\mathsf{b}_1\mathsf{b}_2)$$
$$\boldsymbol{\eta}_q[5] = 1 \cdot g_q(\mathsf{d})$$
$$\boldsymbol{\eta}_q[6] = \boldsymbol{\eta}_q[1] \cdot g_q((\mathsf{xy})^{\mathsf{d}})$$
$$\boldsymbol{\eta}_q[7] = 1 \cdot g_q(\mathsf{h})$$
$$\boldsymbol{\eta}_q[8] = \boldsymbol{\eta}_q[7] \cdot g_q(\mathsf{f})$$
$$\boldsymbol{\eta}_q[9] = \boldsymbol{\eta}_q[7] \cdot g_q(\mathsf{g}_1\mathsf{g}_2)$$

The second invocation computed and stored the tree representation in the global arrays $\boldsymbol{\rho}_q$, and $\boldsymbol{\varphi}_q$:

$$\begin{array}{ll}
\boldsymbol{\rho}_q[1]{=}0 & \boldsymbol{\varphi}_q[1]{=}\{\alpha(\mathsf{e}_1), \alpha(\mathsf{e}_2)\} \\
\boldsymbol{\rho}_q[2]{=}1 & \boldsymbol{\varphi}_q[2]{=}\{\alpha(\mathsf{c}_1), \alpha(\mathsf{c}_2)\} \\
\boldsymbol{\rho}_q[3]{=}2 & \boldsymbol{\varphi}_q[3]{=}\{\alpha(\mathsf{a}_1), \alpha(\mathsf{a}_2), \alpha(\mathsf{a}_2), \alpha(\mathsf{a}_3)\} \\
\boldsymbol{\rho}_q[4]{=}2 & \boldsymbol{\varphi}_q[4]{=}\{\alpha(\mathsf{b}_1), \alpha(\mathsf{b}_2)\} \\
\boldsymbol{\rho}_q[5]{=}0 & \boldsymbol{\varphi}_q[5]{=}\{\alpha(\mathsf{d})\} \\
\boldsymbol{\rho}_q[6]{=}1 & \boldsymbol{\varphi}_q[6]{=}\{\alpha(\mathsf{x}, \boldsymbol{\eta}_q[5]), \alpha(\mathsf{y}, \boldsymbol{\eta}_q[5])\} \\
\boldsymbol{\rho}_q[7]{=}0 & \boldsymbol{\varphi}_q[7]{=}\{\alpha(\mathsf{h})\} \\
\boldsymbol{\rho}_q[8]{=}7 & \boldsymbol{\varphi}_q[8]{=}\{\alpha(\mathsf{f})\} \\
\boldsymbol{\rho}_q[9]{=}7 & \boldsymbol{\varphi}_q[9]{=}\{\alpha(\mathsf{g}_1), \alpha(\mathsf{g}_2)\}
\end{array}$$

In general, the invocation $V_q(e, r)$ generates the Gödel tree of expression $e^{e'}$, where r is the root of the Gödel sub-tree of e', which must have been computed previously. If e is a product of several sub-expressions, i.e., $e = e_1 \times \cdots \times e_k$, parenthesized arbitrarily, then the recursive calls to V_q will follow this parenthesization pattern, but will not generate any nodes for it. Instead, V_q will accumulate the multi-set of primes returned by the recursive call on all e_i, $i = 1, \ldots, k$.

A Gödel tree node is created only when the recursive run encounters an exponentiation operator. Line 9 creates the node, and line 10 fills its data.

The Gödel tree can be thought of as an extended trace of the run of F_q, in the sense that for each node u, integer $\boldsymbol{\eta}_q[u]$ is the product modulo q of primes stored in $\boldsymbol{\varphi}_q[u]$ and in all of its ancestors (See Definition 5.1).

Lemma 5.4. After line 9 of Algorithm 3, $\boldsymbol{\eta}_q[u] = \pi_q(u) \bmod q$ for every node $u = 1, \ldots, v$.

PROOF. Trivial by induction on u. (The inductive step compares line 10 in functions F_q and V_q using the similarity of the stack logs of lines 5 and 9 in Algorithm 3.) $\square$

As a result of the uniqueness of the factorization of integers into primes, integer $\pi_q(u)$ *uniquely* encodes the entire multi-set $\boldsymbol{\phi}_q(u)$. We would therefore expect that the temporary values

$$\boldsymbol{\eta}[0], \boldsymbol{\eta}[1], \ldots$$

computed by the *non*-hashed Gödel encoding, will be precisely

$$\pi_q(0), \pi_q(1), \ldots$$

of the multi-sets

$$\boldsymbol{\phi}_q(0), \boldsymbol{\phi}_q(1), \ldots$$

as computed by V_q. In the same fashion, one may try to rely on the fact V_q *accumulates* the primes generated by the encoding, rather than multiplying these modulo q, to infer that the Gödel tree it generates will not depend at all on q.

These conclusions are indeed true, but only when q is collision free. To understand why, recall that function V_q searches the hash-table, and that these searches use as keys the values in the array $\boldsymbol{\eta}_q[0, \ldots, n]$ as generated by F_q. Function F_q, in turn makes an extensive use of q. If q is colliding, then two distinct temporary values $\boldsymbol{\eta}_q[u_1]$ and $\boldsymbol{\eta}_q[u_2]$ become equal by the modulo q operation in function F_q.

Thus, we have that the multi-sets $\boldsymbol{\phi}_q(u)$ do not depend on q as long as it is collision free. In this case, two such multi-sets are equal precisely when the expressions they represent are equal.

The next lemma gives us a test for checking whether q is colliding.

Lemma 5.5. If q is colliding, then there exist nodes u_1 and u_2 such that the following two conditions holds

$$\begin{aligned} \boldsymbol{\phi}_q(u_1) &\neq \boldsymbol{\phi}_q(u_2), \\ \boldsymbol{\eta}_q[u_1] &= \boldsymbol{\eta}_q[u_2]. \end{aligned}$$

PROOF. Examine the first point in the course of calculation in which a hash collision occurs. Specifically, for all colliding pairs $\boldsymbol{\eta}[u_1] \neq \boldsymbol{\eta}[u_2]$ pick the one with minimal u_2. Since this is the first collision, we have that $\boldsymbol{\eta}[u_1] = \pi_q(u_1)$ is distinct from $\boldsymbol{\eta}[u_2] = \pi_q(u_2)$, thus $\boldsymbol{\phi}_q(u_1) \neq \boldsymbol{\phi}_q(u_2)$. However, from Lemma 4.5, it follows that

$$\begin{aligned} \boldsymbol{\eta}_q[u_1] &= \boldsymbol{\eta}[u_1] \bmod q, \\ \boldsymbol{\eta}_q[u_2] &= \boldsymbol{\eta}[u_2] \bmod q, \end{aligned}$$

thus $\boldsymbol{\eta}_q[u_1] = \boldsymbol{\eta}_q[u_2]$. $\square$

Therefore the condition in line 10 assures us that q is collision-free when we exit the loop. This condition can be checked in $O(n \log n)$ time and $O(n)$ space [Zibin et al., 2003, Section 5].

In the unlikely event that q is colliding then Algorithm 3 reiterates the main loop to select a new prime q. Since the probability of this event is $O(\log n/n)$ (Lemma 4.8), the expected number of iterations is $1 + o(1)$. We therefore have:

Theorem 5.6. Algorithm 3 determines (after a pre-processing stage, depending solely on n) whether $\epsilon_1 = \epsilon_2$ in $O(n)$ space and $O(n \log n)$ expected time.

6. Conclusions and Open Problems

In testing formal equality of ME-expressions, we substitute *non-random* values, specifically primes, for input variables, and rely on unique prime factorization to show equality of products. We circumvent the problem of raising an expression to a non-constant power by using an assignment function, which chooses a new prime for each base and exponent pair. This is possible using an implicit normal form, in which all bases are primitive variables. Although a detailed description of this normal form and a linear tree-structured representation of it appear elsewhere [Zibin et al., 2003], the reader may gain some intuition by examining (5.1) together with Figure 5.1.

The Monte-Carlo algorithm implicitly follows the normal form to compute the hashed Gödel encoding of the inputs, where the hashing is by computing products modulo a polynomially large prime. The Las-Vegas algorithm is in essence a reiteration of the Monte-Carlo algorithm, in which the Gödel tree (structured much like the normal form) is computed. Checking whether no collisions occurred during the hashing can be done in $O(n \log n)$ time using this tree, but the details are described elsewhere [Zibin et al., 2003, Section 5].

An interesting open problem is to improve the *non*-polynomial algorithm [Gurevič, 1985] for determining equality of expressions involving additions, multiplications *and* exponentiation. This paper showed how to efficiently determine equality of ME-expressions (involving multiplication and exponentiation), while equality of polynomials (involving additions and multiplications) has a well known $O(n \log n)$ time randomized algorithm (e.g., see [Chen and Kao, 1997]). Another interesting open problem is to find other applications of the technique of replacing exponentiation by hashing.

References

Agrawal, M., Kayal, N., and Saxena, N. (2002). PRIMES is in P. Technical report, Indian Institute of Technology, Kanpur.

Bruce, K. B., Di Cosmo, R., and Longo, G. (1991). Provable isomorphisms of types. *Math. Structures in Comp. Sci.*, 1:1–20.

Chen, Z.-Z. and Kao, M.-Y. (1997). Reducing randomness via irrational numbers. In *Proceedings of the 29th annual ACM symposium on Theory of computing*, pages 200–209. ACM Press.

Considine, J. (2000). Deciding isomorphisms of simple types in polynomial time. Technical report, Computer Sciences Department, Boston University.

Di Cosmo, R. (1995). *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser. ISBN-0-8176-3763-X.

Dietzfelbinger, M., Karlin, A. R., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. (1994). Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761.

Gurevič, R. (1985). Equational theory of positive numbers with exponentiation. *American Mathmatical Society*, 94(1):135–141.

Pritchard, P. (1981). A sublinear additive sieve for finding prime number. *Communications of the ACM*, 24(1):18–23.

Rittri, M. (1990). Retrieving library identifiers via equational matching of types. In Stickel, M. E., editor, *Proc. of the 10th International Conference on Automated Deduction (CADE'90)*, volume 449 of *Lecture Notes in Computer Science*, pages 603–617, Kaiserslautern, Germany. Springer Verlag.

Schwartz, J. (1980). Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717.

Soloviev, S. V. (1983). The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400.

Tarski, A. (1951). *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA, 2nd edition.

Zibin, Y., Gil, J., and Considine, J. (2003). Efficient algorithms for isomorphisms of simple types. In *Proc. of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 160–171, New Orleans, Louisiana, USA. ACM Press, New York, NY, USA.