FIRST-ORDER TERM COMPRESSION: TECHNIQUES AND APPLICATIONS

JAMES R. CHENEY
CARNEGIE MELLON UNIVERSITY

Master's Thesis in Mathematics
Advisor: Peter Lee

Abstract. Lossless sequential data compression techniques are well-known. Such techniques work best on data with repetition in its sequential representation. One might wish to compress data that can possess non-sequential repetitive structure, for example, first-order terms. Sequential techniques often do not compress such data well. This paper presents improved compression techniques that can take advantage of repetitive term structure. The first part presents a formal background for the problem of compressing terms. The second part outlines several approaches to compressing terms by pattern substitution, using a modified form of substitution tree indexing. The third part shows how term compression can be used to compress proofs in a system for mobile code safety and presents experimental results establishing the effectiveness of term compression.

Contents

Introduction

1. Background

General lossless data-compression algorithms[14] try to find more compact representations for objects, from which the original objects can be reconstructed exactly. Many such algorithms compress data by searching for more efficient encodings that take advantage of repetition in the data. For example, in compressing English text, some words recur so often that if they were replaced by smaller "nonsense words", the text would become much shorter. Of course, such a text would be unreadable, and it would be necessary to decode it before reading it. Some such abbreviations are so common that their meanings are universally known: *e. g.*, *i. e.*, *etc.* Compressing text by making up new words that have meaning only for a particular encoding is an example of *dynamic* encoding; using well-known conventional abbreviations is an example of *static* encoding. This thesis will concentrate on dynamic compression methods, though it is not difficult to modify the algorithms presented here for use in static compression.

Statistical encoding is a compression technique that uses statistics and information theory to efficiently encode data. One example of statistical encoding is *Huffman encoding*[16]. In Huffman encoding, sequences are encoded by replacing the symbols which occur most frequently with short sequences of bits, and infrequent symbols with longer sequences. To obtain the original data, the encoded data must be decoded using the correspondence between symbols and bit sequences (usually represented as a *Huffman tree*). In most cases the space savings resulting from using short abbreviations for frequently-occurring symbols outweigh the extra space required for the infrequent symbols and for the encoding. Huffman encoding is truly general because it makes no assumptions on the structure of the data. But by the same token, Huffman encoding cannot take advantage of such structure where it exists, so it may be possible to get better results by using knowledge of structure.

If the input data, viewed as a sequence of symbols, contains sequential repetition, then the data can be compressed using *pattern substitution*. In this approach, the input sequence is searched for *repeated subsequences* or *patterns* of symbols. These subsequences are encoded by new symbols in a *pattern dictionary*, and the input is encoded using new symbols from this dictionary. As in statistical encoding, to reconstruct the original data, the compressed form must be decoded using the dictionary. In this approach, space savings occur if the dictionary together with the new representation of the input is smaller than the original input. Because the search space of possible dictionaries and encodings is exponential in the size of the input sequence (we will provide a proof in Section 1.3), it is not worthwhile to search for *optimal* compressions in general; instead, many useful *approximate* algorithms exist, some of which asymptotically approach optimality (such as the *Lempel-Ziv* algorithm [26]). Some approximate algorithms are good at general data compression, while others perform very well on special kinds of data but not so well in general. Sequential compression methods are less general than Huffman compression, but are capable of better performance because they can take advantage of sequential repetition where it exists.

Many kinds of data might provide opportunities for compression based on redundant non-sequential structure. Objects such as formal proofs, syntax trees, and machine code have structure which is not strictly linear, but which could contain repeated patterns. All objects stored in machine-readable form can be expressed as sequences of bits or characters, so sequential compression methods can be used on non-sequential data. However, non-sequential patterns which are actually present in a structure may be hidden in a linearization of the structure, so this approach may miss opportunities for compression. So non-sequential data often cannot be compressed as effectively as sequentially repetitive data such as natural-language text, source code, and graphical bitmaps.

By analogy with the case for sequential data compression, it seems reasonable to expect that more advanced techniques be better able to compress terms with non-sequential structure. In this thesis, we present techniques for compressing first order terms.

2. Overview

In Part 1, I will provide the mathematical foundations which will be used in the rest of the thesis, including definitions and properties relating to terms and term compression. Operator domains, first-order terms, term patterns, substitutions, and term rewriting are standard concepts important to our discussion. Some additional definitions and concepts that are of use in this thesis will also be given. Where appropriate for our purposes, propositions relating these concepts will be proved.

In Part 2 of this thesis, I will study the general problem of compressing terms. I will provide an abstract framework in which to express the term compression problem, and show how to express the sequential compression problem as a special case. I will argue that, as in the sequential case, general optimal compression is difficult, but there are acceptable approximate algorithms, and I will outline several approaches that may be practical.

In Part 3, I will show how term compression techniques can be applied to a practical problem in program analysis. A recent proposed system for mobile code safety, *proof-carrying code* [18], involves transmitting programs together with safety proofs over a network. Because proofs in a current implementation can be significantly (in principle, exponentially) larger than the code they accompany, it is important to find more efficient representations of the proofs. Proofs in this system are terms in an appropriate operator domain, so they can be compressed using the techniques from Part 2 on the corresponding terms.

Part 1. Definitions and properties

1. Basic Definitions

1.1. Preliminaries.

Definition 1.1.1. We write A^n for the set of all n-tuples of elements of A, and $A^{(\mathbb{N})}$ for $\bigcup_{i\geq 0} A^i$.

The *power set* of S, denoted $\mathcal{P}(S)$, is the set of all subsets of S.

We will use the standard notation for functions, so that $f : A \to B$ means that $f \subseteq A \times B$ and for every $x \in A$ there is exactly one y such that $(x, y) \in f$, or alternately $f(x) = y$. Given a function $f : A \to B$, we will denote the domain of f by $Dom(f) = A$ and the range of f by $Rng(f) \subseteq B$. A partial function which maps a subset of A to B is denoted $f : Dom(f) \subseteq A \to B$.

We will denote the number of times an element e occurs in a multiset M by $\mu_M(e)$, with $\mu_M(e) = 0$ meaning $e \notin M$. We will not distinguish notationally between sets and multisets, instead considering sets to be multisets such that for any set S and potential element e, $\mu_M(e) \in \{0, 1\}$.

We will make use of *rooted trees* T, consisting of a set V (or $V(T)$) of *vertices*, E (or $E(T)$) $\subseteq V \times V$ of *edges*, and a distinguished node r (or $r(T)$) $\in V$ called the *root*. If v is the parent of w then the edge connecting them is (v, w).

1.2. Terms.

Definition 1.2.1. An *operator domain* is a pair (Σ, ϕ) where $\phi : \Sigma \to \mathbb{N}$ is called the *rank function*. An element $\sigma \in \Sigma$ is called an *operator*. An operator σ is said to have *rank* $n = \phi(\sigma)$. We define the partition $\{\Sigma_n \mid n \geq 0\}$ of Σ by $\Sigma_n = \{\sigma \in \Sigma \mid \phi(\sigma) = n\}$. An element of Σ_0 is also called a *constant*.

We may abbreviate (Σ, ϕ) by Σ when the ranks of the elements of Σ are clear from context or not important.

Given a set X, disjoint from Σ, called the set of *variables*, *(ΣX-)terms* are recursively defined as follows:

1. For each $x \in X$, x is a term.
2. For each $\sigma \in \Sigma_0$, σ is a term.
3. For each $\sigma \in \Sigma_n$, for each n-tuple of terms $(t_1, \dots, t_n)$, $\sigma(t_1, \dots, t_n)$ is a term.

We denote the set of all ΣX-terms by $Term_{\Sigma X}$. The variables which occur in a term t are called the *free variables* of t, denoted $Fr(t)$. A term with no free variables is called *closed*. The set of operators which occur in a term t is here denoted $Op(t)$.

If (Σ, ϕ) is an operator domain and X is a set of variables, then we define the *X-invariant operator domain* to be the operator domain $(\Sigma \cup X, \phi_X)$, where $\phi_X(x) = 0$ if $x \in X$, and $\phi_X(\sigma) = \phi(\sigma)$ if $\sigma \in \Sigma$. For convenience we may refer to variables in terms as operators with rank 0 without explicitly appealing to this definition.

Remark 1.2.2. Note that variables in terms as we have defined them may only appear as operands; that is, they may only be used to stand for constants, never for operators with arguments. Consequently our term system can represent only first-order systems. This is also the reason why we define $\phi_X(x) = 0$ for $x \in X$, in invariant operator domains.

1.3. Patterns.

Definition 1.3.1. Given a (typically infinite) set Y of variables distinct from X, we will define *patterns* to be $(\Sigma \cup X)Y$-terms. The set of all such patterns is denoted

$Pat^Y_{\Sigma X}$. The elements $\mathbf{y} \in Y$ are called *pattern variables*, and the elements $x \in X$ are called *term variables*.

A pattern of the form $\mathbf{y}$ where $\mathbf{y} \in Y$ is called *trivial*.

A pattern of the form σ or $\sigma(\mathbf{y}_1, \dots, \mathbf{y}_n)$ for $\sigma \in \Sigma \cup X$ and $\mathbf{y}_1, \dots, \mathbf{y}_n \in Y$ is called *atomic*.

A pattern with the property that each free pattern variable in it occurs exactly once is called *linear*.

Remark 1.3.2. The reason for using patterns and pattern variables is to eliminate any possibility of confusion between variables that already exist in a term we wish to compress, and variables we introduce in the process of discovering repeated term structures. We will later speak of substitutions, matching, and generalization only with respect to patterns, even though these concepts can be defined just as easily for terms.

The relation between pattern variables and term variables is similar to the relation of meta-variables to ordinary variables in any mathematical study of terms. In our case, however, not only terms but also term forms containing (pattern) meta-variables are the objects of formal study. Fortunately, we will not require a sufficiently formal meta-theory to require an additional layer of meta-meta-variables.

1.4. **Substitutions.**

Definition 1.4.1. For an arbitrary variable set $Z \subset Y$ and operator domain (Σ, ϕ) over variable set X, a Z*-substitution* is a function $s : Z \to Pat^Y_{\Sigma X}$. The set of all possible Z-substitutions is called $Sub^Z_{\Sigma X}$.

Given a substitution

$$s = \{(\mathbf{z}_1, p_1), \dots, (\mathbf{z}_n, p_n)\},$$

we will also use the notation

$$s = \mathbf{z}_1 \mapsto t_1, \dots, \mathbf{z}_n \mapsto t_n.$$

The *empty substitution* is the single element $\emptyset \in Sub^{\emptyset}_{\Sigma X}$, also denoted by $\cdot$.

For any Z-substitution s and pattern p, the pattern p' identical to p except that every free pattern variable $\mathbf{y}$ occurring in p has been replaced with $s(\mathbf{y})$ is called the result of *applying* s to p. We also denote this pattern by $[s]p$.

A substitution s is *idempotent* if $[s]p = [s][s]p$ for every $p \in Pat^Y_{\Sigma X}$.

A substitution is *minimal* if $(\mathbf{y}, \mathbf{y}) \notin s$ for every $\mathbf{y} \in Y$.

Remark 1.4.2. Note that s is idempotent only if

$$Z \cap \bigcup_{\mathbf{z} \in Z} Fr(s(\mathbf{z})) = \emptyset. \tag{1}$$

An idempotent substitution is also minimal, and conversely if a substitution is minimal and satisfies (1) then it is idempotent. So, a substitution is idempotent if and only if it is minimal and (1) holds.

Definition 1.4.3. An *invertible* substitution $s \in Sub^Z_{\Sigma X}$ is a substitution which is also a bijection from Z to Z. The substitution s^{-1} is called the *inverse* of s. Invertible substitutions are also called *renamings*, and the result of applying a renaming to a term is a *renaming* of the term.

Remark 1.4.4. It is customary to assume that appropriate renamings can always be chosen to avoid unintentional conflicts between variable names in terms.

1.5. **Pattern matching and generalization.**

Definition 1.5.1. A substitution s *matches* pattern p to pattern q if $q = [s]p$. If there is such a substitution, we say p matches q under s.

The weak pattern ordering $\preceq$ is defined so that $p \preceq q$ means that p matches q, for $p, q \in Pat^Y_{\Sigma X}$. Note that $\preceq$ is reflexive and transitive.

If $p \preceq q$ and $q \preceq p$ then we say that $p \equiv q$, or p is equivalent to q. Observe that $\equiv$ is reflexive, transitive, and symmetric. Observe also that $p \equiv q$ if and only if the substitution matching p to q is invertible; its inverse matches q to p. If $p \preceq q$ and $p \not\equiv q$, then we write $p \prec q$.

There may be many substitutions which match a pattern to another. Suppose substitutions $s_1 \in Sub^Z_{\Sigma X}$ and $s_2 \in Sub^{Z'}_{\Sigma X}$ both match pattern p to q. We say that s_1 is *more general* than s_2 if $Z \subseteq Z'$ and for each $\mathbf{z} \in Z$, $s_1(z) \preceq s_2(z)$. A matching substitution s is *most general* if for every matching substitution t, $s \preceq t$.

Remark 1.5.2. Matching is a special case of *unification*, the problem of finding a substitution s for two patterns p and q such that $[s]p = [s]q$. There are known efficient algorithms for finding most general unifiers[2, 20]. Matching can be implemented in $O(n)$ time, where n is the size of the input patterns or terms.

Remark 1.5.3. Observe that a most general matching substitution never contains $\mathbf{y} \mapsto \mathbf{y}$ for any $\mathbf{y} \in Y$, since if there were such a substitution s, there would be another matching substitution $s \backslash \{(\mathbf{y}, \mathbf{y})\}$, and this substitution would be more general than s. Therefore, all most general matching substitutions are minimal.

Definition 1.5.4. A pattern q is a *generalization* of patterns p_1 and p_2 if there are substitutions s_1 and s_2 such that $p_1 = [s_1]q$ and $p_2 = [s_2]q$.

Patterns may also have more than one generalization. Suppose both q_1 and q_2 generalize patterns p_1 and p_2. We say that q_1 is *more specific* than q_2 if $q_2 \prec q_1$. If for every other generalization q, $p \preceq q$, then p is *most specific*.

Remark 1.5.5. Generalization is also sometimes called *anti-unification*, since it is the dual operation to unification. There are known efficient algorithms for finding most specific anti-unifiers [2, 17]. Generalization is $O(n^2)$ if n is the size of the input patterns.

1.6. **Substitution matching and generalization.**

Definition 1.6.1. The *composition* of substitution $s \in Sub^Z_{\Sigma X}$ with substitution $t \in Sub^{Z'}_{\Sigma X}$, is the substitution $s \circ t \in Sub^{Z \cup Z'}_{\Sigma X}$ defined so that

$$(s \circ t)(\mathbf{y}) = \begin{cases} [s](t(\mathbf{y})) & \text{if } \mathbf{y} \in Z' \\ s(\mathbf{y}) & \text{if } \mathbf{y} \in Z \backslash Z'. \end{cases}$$

A substitution t *matches* s_1 to s_2 if $t \circ s_1 = s_2$. If s_1 and s_2 match then we write $s_1 \preceq s_2$.

If $s \preceq t$ and $t \preceq s$ then we write $s \equiv t$, or s is equivalent to t. Again note that then the matching substitutions are invertible and inverses of each other. Observe that $\cdot$ is an invertible substitution which is its own inverse and that every invertible substitution s satisfies $s \equiv \cdot$, since $\cdot \preceq s$ $(s \circ \cdot = s)$ and $s \preceq \cdot$ $(s^{-1} \circ s = \cdot)$. We will therefore write $s \equiv \cdot$ for "s is invertible".

We also define the concepts of *more general* and *most general* matching substitutions analogously to the definitions for pattern matching.

A substitution t is a *generalization* of two substitutions s_1 and s_2 under substitutions u_1 and u_2 if $s_1 = u_1 \circ t$ and $s_2 = u_2 \circ t$.

More specific and *most specific* generalizations of substitutions are defined analogously to the concepts for pattern generalization.

We say that s is more general (or more specific) than t if $s \prec t$ (or $t \prec s$). Observe that the concept "more general" defined here is the same as that of Section 1.5 if restricted to matching substitutions.

2. Additional Concepts

Here we introduce some less familiar ideas that extend the basic ideas discussed above in ways that will prove helpful to our discussion. Specifically, we will introduce analogoues to patterns and substitutions using term-position notation, and show how to use these concepts to define specific rewritings of terms. Some of these ideas are drawn from [4, 3].

2.1. Term positions and replacements.

Definition 2.1.1. We define a set Pos of *term positions*. The set of positions present in a term t is denoted $Pos(t)$. We also define functions $oper : Term_{\Sigma X} \times Pos \to \Sigma \cup X$ and $subterm : Term_{\Sigma X} \times Pos \to Term_{\Sigma X}$ such that $oper(t,p)$ is the operand or variable that occupies position p in t and $subterm(t,p)$ is the subterm starting at position p in t.

Positions may be represented as sequences of nonzero integers (in Dewey decimal notation), with the empty sequence denoting the root position, and the sequence $(i_1, \ldots, i_n)$ denoting the position found by following the i_jth subposition of each position starting at the root.

The relation $\leq_P = \{(l,m) \mid l \text{ is a prefix of } m\}$ is an ordering on Pos. The strict relation $<_P$ is defined so that if $l <_P m$ then $l \neq m$ and $l \leq_P m$.

For convenience, we will refer to $oper(t,())$ as the root operator of t.

Example 2.1.2. Consider the term $t = \sigma(\gamma(a,b),c)$. The position $()$ denotes the entire term; (1) denotes $\gamma(a,b)$, (2) denotes c, and $(1,2)$ denotes b. The set $Pos(t) = \{().(1),(1,1),(1,2),(2)\}$.

Definition 2.1.3. Given a term s and position p in term t, the term identical to t except that the subterm of t at p is replaced by s is denoted $\langle s\rangle_p t$. A term u identical to t everywhere except possibly at or below p is called a *context* for replacing the subterm at p in t.

Example 2.1.4. Consider again $t = \sigma(\gamma(a,b),c)$, and also $u = \gamma(x,y)$ and $v = \sigma(d,c)$. We then have $\langle u\rangle_{(1,2)} t = \sigma(\gamma(a,\gamma(x,y)),c)$ and $\langle u\rangle_{(1)} t = \sigma(\gamma(x,y),c)$. Also, v is a context for the latter replacement at (1) in t. Note that $\langle u\rangle_{(1)} v = \langle u\rangle_{(1)} t$.

Remark 2.1.5. If u is a context for a replacement at p in t, then for any term s, $\langle s\rangle_p u = \langle s\rangle_p t$.

2.2. Term Rewriting Systems.

Definition 2.2.1. A *rewrite relation* is a relation $\to$ over $Pat^Y_{\Sigma X}$ which is:

1. closed under substitution: if $q \to r$ then $[s]q \to [s]r$, for any $s \in Sub^{Z \subseteq Y}_{\Sigma X}$.
2. closed under context replacement: if $q \to r$ then $\langle u\rangle_p q \to \langle u\rangle_p r$ for any $u \in Pat^Y_{\Sigma X}$ and $p \in Pos(u)$.

Definition 2.2.2. A *rewrite system* is a relation R over $Pat^Y_{\Sigma X}$ such that for each $(p, q) \in R$, $Fr(q) \subseteq Fr(p)$. The rewrite relation $\to_R$ is the $\subseteq$-minimal rewrite relation containing R. Term s is said to *rewrite* to t if $s \to_R t$.

Definition 2.2.3. A rewrite system R is *terminating* if there is no infinite sequence of terms t_i $(i \in \mathbb{N})$ with $t_0 \to_R t_1 \to_R t_2 \cdots$.

A rewrite system R is *confluent* if whenever $t \to_R u$ and $t \to_R v$, there exists an x such that $u \to_R x$ and $v \to_R x$.

A rewrite system is *complete* if it is terminating and confluent.

We say that a rewrite system is *decreasing* with respect to $f : Term_{\Sigma X} \to \mathbb{N}$ if for every s, t such that $s \to_R t$, $f(t) < f(s)$.

A rewrite system is *left-linear* if for each $(p, q) \in R$, p is linear. A rewrite system is *left-atomic* if for each $(p, q) \in R$, p is atomic.

A left-linear rewrite system R is *orthogonal* if no left-hand side of a rule unifies with a renamed non-variable subterm (other than itself) of any rule's left-hand side. Two patterns p_1 and p_2 *unify* if there is a substitution s such that $[s]p_1 = [s]p_2$. Formally, R is orthogonal if for every rule $r = (p, q)$, for every left-hand size of a rule $s = (p', q')$, for every position $l \in Pos(p')$ with $oper(p', l) \notin X$, if $r \neq r'$ or $l \neq ()$ then for every renaming t and substitution s, $[s]p \neq [s]([t]p')$.

The relation $R^{-1} = \{(q, p) \mid (p, q) \in R\}$ is the *reverse* of R. We denote $\to_{R^{-1}}$ by $\leftarrow_R$. A rewrite system R is *reversible* if R^{-1} is a rewrite system.

We say that a rewrite system R is *orderable* if there is an ordering of the rules of R such that for each i, every operator occurring in the ith rule's left-hand side occurs only in that pattern and in the right-hand sides of later rules.

If $t \to_R u$ implies $t = u$ then t is in *R-normal form*. If $s \to_R t$ and t is in normal form then t is a R-normal form (or normalization) of s.

Lemma 2.2.4. *A rewrite system R is reversible if and only if $Fr(p) = Fr(q)$ for each $(p, q) \in R$, and if R is reversible then R^{-1} is reversible and ${R^{-1}}^{-1} = R$.*

Proof. Trivial. ☐

Lemma 2.2.5. *Decreasing systems are terminating.*

Proof. If there were a system decreasing with respect to some f containing a non-terminating derivation $t_0 \to_R t_i \to_R \cdots$, then $f(t_0) \geq f(t_1) \geq \cdots$ would constitute an infinite decreasing sequence of $\mathbb{N}$. ☐

Lemma 2.2.6. *If a rewrite system R is left-atomic and left-linear, and its rules have distinct left-hand root operators (that is, no two left-hand sides*

$$\sigma(\mathbf{y}_1, \dots, \mathbf{y}_n),\ \sigma'(\mathbf{z}_1, \dots, \mathbf{z}_m)$$

are such that $\sigma = \sigma'$), then R is orthogonal.

Proof. The only nontrivial subterm of an atomic term is the term itself. No two terms with different root operators can be unified. Since R is left-linear, it must be orthogonal. ☐

Theorem 2.2.7 (Rosen [23])**.** *Orthogonal systems are confluent.*

Lemma 2.2.8. *Finite ordered systems are terminating.*

Proof. We proceed by induction on the number of rules. Suppose that there are no rules. The empty system is ordered and it terminates. Suppose that every ordered system with fewer than n rules terminates. Let R be an ordered rewrite system with n rules $(r_1, \ldots, r_n)$ in order. Put $(p, q) = r_n$. Assume that $t_0 \to_R t_1 \to_R \cdots$ is an infinite derivation of a pattern t. If there were finitely many uses of r_n, then for some m, $t_m \to_R t_{m+1} \to_R \cdots$ would be an infinite derivation using only the first $n-1$ rules, contradicting the induction hypothesis. Therefore there must be infinitely many uses of r_n in the infinite derivation. However, there can be at most finitely many occurrences of p in t, and no more can be introduced by the earlier rules because they cannot contain any of the operators in p in their right-hand sides. Consequently, there cannot be infinitely many uses of the last rule in any derivation. Therefore it is impossible for any term to have an infinite derivation, so the R terminates. Consequently by induction, every finite ordered system terminates. □

Lemma 2.2.9. *If R is complete then every term s has exactly one R-normal form.*

Proof. Since R is complete, it terminates and is confluent. Clearly s has at least one normal form because R terminates. Suppose $s \twoheadrightarrow_R t$ and $s \twoheadrightarrow_R u$ and both are in normal form. By confluence, there exists x such that $t \twoheadrightarrow_R x$ and $u \twoheadrightarrow_R x$. Then $t = x = u$. □

Theorem 2.2.10.
1. *If R converges then every term s and t with $t \twoheadleftarrow_R s$ have the same R-normal form.*
2. *If s is in R-normal form and $t \twoheadleftarrow_R s$ and R converges then s is the R-normal form of t.*

Proof.
1. Suppose x is an R-normal form of t. Clearly $s \twoheadrightarrow_R t \twoheadrightarrow_R x$, so x is also an R-normal form for s. Because R converges, by 2.2.9 x is the unique normal form for both s and t.
2. A corollary of 1.

□

Remark 2.2.11. There is, of course, much more to rewrite systems than this, but we will only need to study simple rewriting systems for which no advanced discussion is required. The above lemmas suffice to show that the rewrite systems we will use to encode terms terminate, and that those we use to decode terms converge, and moreover that decoding any encoding of a term restores the term.

2.3. **Positions and substitutions.**

Definition 2.3.1. Given a term or pattern t, a *(positional) substitution* with respect to t over variable set Z is a relation $s \subseteq Z \times Pos(t)$ such that
1. $subterm(t, l) = subterm(t, l')$ for every $l, l' \in Pos(t)$ and $\mathbf{z} \in Z$ with $(\mathbf{z}, l) \in s$ and $(\mathbf{z}, l') \in s$
2. for every $\mathbf{z} \in Z$, there exists a $l \in Pos(t)$ such that $(\mathbf{z}, l) \in s$.

The set of positional substitutions with respect to t over Z is denoted $PSub_t^Z$. As for variable substitutions, we may use the notation

$$s = \mathbf{z}_1 \mapsto l_1, \ldots, \mathbf{z}_n \mapsto l_n$$

for

$$s = \{(\mathbf{z}_1, l_1), \ldots, (\mathbf{z}_n, l_n)\}.$$

Given a positional substitution $s \in PSub_t^Z$ and $\mathbf{z} \in Z$, the set $\{subterm(t,l) \mid (\mathbf{z},l) \in s\}$ has only one member, which we will denote by $s_t(\mathbf{z})$.

The result of applying a positional substitution to a pattern p is a pattern p' identical to p except that every occurrence of a variable $\mathbf{z} \in Z$ in p has been replaced by $s_t(\mathbf{z})$. We also denote application by $[s]_t p$.

Example 2.3.2. Consider $t = \sigma(\sigma(\gamma(x,a),y),\gamma(x,a))$. Suppose $s = \mathbf{y} \mapsto (1), \mathbf{z} \mapsto (2)$. Then $s_t(\mathbf{y}) = \sigma(\gamma(x,a),y)$ and $s_t(\mathbf{z}) = \gamma(x,a)$. Clearly s is a valid positional substitution.

Suppose $s' = \mathbf{y} \mapsto (1,1), \mathbf{y} \mapsto (2)$. Then $s_t(\mathbf{y}) = \gamma(x,a)$ is well defined, so s' is also a valid positional substitution.

Consider $p = \gamma(\mathbf{y},\mathbf{z})$. Then

$$[s]_t p = \gamma(\sigma(\gamma(x,a),y),\gamma(x,a))$$

and

$$[s']_t p = \gamma(\gamma(x,a),\mathbf{z})$$

.

Definition 2.3.3. We say that a pattern p *matches* a term t at l if there is a positional substitution s such that $[s]_t p = subterm(t,l)$. The *multiplicity* of a pattern in a term is the number of positions in the term at which the pattern matches the term.

We also say that a substitution s *matches* a positional substitution u if there is a positional substitution v such that $[v]_t s(\mathbf{z}) = u_t(\mathbf{z})$ for each $\mathbf{z} \in Z$. In other words, if substitution s matches positional substitution u under positional substitution v, then for each $\mathbf{z}$ in the domain of v, u matches $s(\mathbf{z})$ and $v_t(\mathbf{z})$.

Example 2.3.4. Consider again t as above. The pattern $p = \sigma(\mathbf{y},\mathbf{z})$ matches t at $()$ under s as above, and at (1) under $\{(\mathbf{y},(1,1)),(\mathbf{z},(1,2))\}$. The pattern $p' = \sigma(\sigma(\mathbf{y},y),\mathbf{y})$ matches t under s' above.

2.4. **Pattern occurrences.**

Definition 2.4.1. A *pattern occurrence* of p in term or pattern t is a pair $(l,s) \in Pos(t) \times PSub_t^{Fr(p)}$ such that p matches t at l under the positional substitution s. For a given position l and substitution s, if there is a pattern p such that (l,s) is an occurrence of p in t, then there is exactly one such pattern and we say p is the pattern occurring at (l,s) in t, or $pat_t(l,s) = p$.

An occurrence (l,s) is said to *cover* the set $cover(l,s) \subseteq Pos$ defined by

$$cover(l,s) = \{m \in Pos \mid l \leq_P m \text{ and for some } (\mathbf{z},n) \in s,\ m <_P n\}$$

Two distinct occurrences (l_1,s_1) and (l_2,s_2) *overlap* if

$$cover(l_1,s_1) \cap cover(l_2,s_2) \neq \emptyset.$$

If two occurrences do not overlap, they are called *independent*.

Remark 2.4.2. Intuitively, a pattern occurrence covers the set of positions which map to non-pattern variables in the correspondence between operators in the pattern and locations in the term.

Example 2.4.3. Recall $t = \sigma(\sigma(\gamma(x,a),y),\gamma(x,a))$. There are two occurrences of the pattern $p = \sigma(\mathbf{y},\mathbf{z})$ in t, namely $o_1 = ((),\mathbf{y} \mapsto (1),\mathbf{z} \mapsto (2))$ and $o_2 = ((1),\mathbf{y} \mapsto (1,1),\mathbf{z} \mapsto (1,2))$. These occurrences cover $\{()\}$ and $\{(1)\}$ respectively, so they are independent. There is also an occurrence of $p' = \sigma(\sigma(\mathbf{y}, y), \mathbf{y})$ at $o_3 = ((1), \mathbf{y} \mapsto (1,1), \mathbf{y} \mapsto (2))$. This occurrence covers $\{(), (1), (1,2)\}$, so it overlaps with each of the occurrences of p.

Definition 2.4.4. Two occurrences $o_1 = (l, t)$ and $o_2 = (m, u)$ are said to be *adjacent* at m if for some $\mathbf{y}$, $(\mathbf{y}, m) \in t$ and *joinable* at m if they are adjacent at m and $pat_t(o_2)$ matches $subterm(t, m')$ for each $(\mathbf{y}, m') \in t$. The result of *joining* o_2 *to* o_1 *at* m is the occurrence $(v, (t \backslash \{(\mathbf{y}, m)\}) \cup u')$ where u' is a positional substitution identical to u except variable names have been changed so as to avoid conflict with those in t. We assume that some systematic procedure for renaming variables is used so that the result of joining two occurrences is always the same.

The result of joining is also denoted $join_m(o_1, o_2)$.

Example 2.4.5. Consider occurrences o_1 and o_2 above. These occurrences are adjacent and joinable at (1), and $join_{(1)}(o_1, o_2) = ((), \mathbf{y}' \mapsto (1,1), \mathbf{z}' \mapsto (1,2), \mathbf{z} \mapsto (2))$. Note that the variables from o_2 have been renamed to avoid conflicting with the variables of o_1.

Remark 2.4.6. If o_1 and o_2 are adjacent at m and are occurrences of p and q in t respectively then they are joinable. Because $o_1 = (l, t)$ is an occurrence of p in t, it matches t at l. Suppose $(\mathbf{y}, m) \in s$. Then for any other position m' with $(\mathbf{y}, m') \in t$, by the definition of occurrence we have $subterm(t, m) = subterm(t, m')$. Clearly $o_2 = (m, u)$ matches t at m, so it also matches t at m' for any such m'. Therefore o_1 and o_2 are joinable at m.

Part 2. Term compression techniques

In this part of the thesis we will consider the general problem of compressing terms by replacing repeated patterns in the data by symbols predefined in a dictionary. We will first describe this problem formally and investigate its complexity and relation to sequence compression. Our approach to term compression works in two stages: *pattern search*, which identifies redundant patterns in the input, and *term encoding*, which uses a rewrite system (called a *dictionary*) to rewrite the input terms to eliminate redundant patterns. We will give algorithms for complete and approximate pattern search and term encoding. Before doing so, we present *pattern substitution trees*, an efficient data structure for storing sets of patterns and discovering shared pattern structure. Pattern substitution trees are based on the substitution trees of [12, 11] used for term indexing in automatic theorem provers.

1. General Term Compression

1.1. Term Compression.

Definition 1.1.1. A *pattern dictionary* over Σ' is a rewrite system

$$D \subseteq Pat^Y_{\Sigma'\emptyset} \times Pat^Y_{\Sigma\cup\Sigma' X}$$

which:

1. is reversible, and its reverse is decreasing with respect to some function f
2. is left-atomic with distinct root operators from Σ'
3. is left-linear
4. is finite
5. is orderable

Remark 1.1.2. The set Σ' is a set of operators disjoint from Σ, which are used to define new, atomic patterns in the dictionary which represent larger, redundant patterns.

Definition 1.1.3. A *term cost metric* is a function $M : Term_{\Sigma X} \to \mathbb{N}$.

An *operator cost function* is a function $c : \Sigma \cup X \to \mathbb{N}$. The term cost metric M_c is the metric defined so that $M_c(\sigma) = c(\sigma)$ for $\sigma \in \Sigma_0 \cup X$, and $M_c(\sigma(t_1, \dots, t_n)) = c(\sigma) + \sum_{i=1}^{n} M_c(t_i)$ if $\sigma \in \Sigma_n$. We call a metric based on an operator cost function *additive*.

A *dictionary cost metric* is a function N which maps pattern dictionaries to $\mathbb{N}$.

Remark 1.1.4. As mentioned above, pattern dictionaries are analogous to string dictionaries that map new symbols to sequences of input symbols in the sequential case. Usually in the sequential case the goal is to optimize the size of the dictionary and compressed input, that is, to use as few symbols as possible. This is often an appropriate goal for term compression as well, but there might be times when a better goal would be to minimize some other function of the data representation or the dictionary. We will express our algorithms in terms of arbitrary measures of term and dictionary cost instead of some fixed concept of size, in order to retain some flexibility. Most of the time, however, we will have additive metrics such as term size in mind.

Example 1.1.5. The function $M_\#$ which counts the number of symbols occurring in a term is a term cost metric. It is also based on the operator cost function c which assigns a cost of 1 to each operator, constant, and variable.

Similarly, the dictionary cost metric $N_{\#}$ sums the number of symbols of all the entries in the dictionary.

Definition 1.1.6. Given a term cost metric M, dictionary cost metric N, and a multiset of ΣX-terms $\{t_1, \dots, t_n\}$, the *term compression problem* is the problem of finding a dictionary D over Σ' and a multiset of $\Sigma \cup \Sigma' X$-terms $\{t'_1, \dots, t'_n\}$ such that $t'_i \leftarrow_D t_i$ for each i, $(1 \leq i \leq n)$, and

$$N(D) + \sum_{i=1}^{n} M(t'_i) < \sum_{i=1}^{n} M(t_i).$$

An *optimal* solution to the term compression problem is a dictionary and a set of encodings such that $\sum_{i=1}^{n} M(T'_i)$ is the smallest among all solutions.

Remark 1.1.7. There are three parts of the compression system: dictionary construction, encoding, and decoding. Dictionary construction consists of a search for appropriate patterns which are repeated in the input. These patterns become the right sides of rules in a dictionary D. Encoding corresponds to $\leftarrow_D$-derivation, with the goal of finding derivations which minimize the M-values of the results. Decoding corresponds to $\rightarrow_D$-normalization.

Dictionary construction is complex and will be studied more closely in the section on complexity, along with encoding. Encoding and decoding have the following properties:

Theorem 1.1.8. *Encoding a ΣX-term with respect to dictionary D terminates.*

Proof. The relation $\leftarrow_D$ built from the reverse of pattern dictionary D is decreasing, so is guaranteed to terminate. □

Theorem 1.1.9. *Decoding any encoding of a ΣX-term t with respect to dictionary D terminates and produces t.*

Proof. According to Lemma 2.2.6 of Part 1, pattern dictionaries are orthogonal, and therefore confluent by Theorem 2.2.7 of Part 1. Since dictionaries are finite and orderable, they terminate. Therefore D is complete. As a result, the $\rightarrow_D$ relation is guaranteed to terminate and produce a unique normal form. Since the term t cannot contain any symbols of Σ', it is in D-normal form, so if $t' \rightarrow_D t$ then the D-normal form of t' is t. This proves that decoding an encoding of a term produces the original term. □

Example 1.1.10. If the pattern $p = \sigma(\sigma(x, \mathbf{y}_1), \mathbf{y}_2)$ occurs many times in a term, we can use a new binary operator $\sigma' \notin \Sigma$. An appropriate dictionary would contain a rewrite rule $\sigma'(\mathbf{y}_1, \mathbf{y}_2) \rightarrow \sigma(\sigma(x, \mathbf{y}_1), \mathbf{y}_2)$. Then the rewrite system D^{-1} could be used to encode the term, replacing occurrences of p with $\sigma'(\mathbf{y}_1, \mathbf{y}_2)$. Finding the D-normal form of an encoded term would replace all occurrences of $\sigma'(\mathbf{y}_1, \mathbf{y}_2)$ with p, producing the original term again.

Remark 1.1.11. To solve the term compression problem practically, we will split the problem into two phases: searching for patterns, and encoding the term with these patterns. Pattern search takes a set of terms as input and yields a pattern multiset, with multiplicities indicating the number of occurrences seen in the term. Term encoding algorithms are given a set of terms and a pattern multiset and produce a pattern dictionary and a set of encoded terms. We will present algorithms for both these phases in the the last two sections of Part 2.

The algorithm for decoding an encoded term using a dictionary is straightforward, so we will not devote any discussion to it.

1.2. **Sequential Compression.** In standard sequential compression algorithms data is represented as a sequence s of symbols in some alphabet Γ. A sequence can be represented as a term, with unary operators σ_γ for each $\gamma \in \Gamma$ and a constant $\epsilon \notin \Gamma$. The constant ϵ represents the empty string, while if s' represents $s \in \Gamma^*$, then $\sigma_\gamma(s')$ represents the string γs. This transformation can also be easily reversed. The following algorithm formalizes this procedure.

Algorithm 1.2.1 (trans).
Given alphabet Γ of symbols, define operator domain (Σ, ϕ) so that $\Sigma = \{\sigma_\gamma \mid \gamma \in \Gamma\} \cup \{\epsilon\}$ where $\epsilon \notin \Gamma$, and $\phi(\sigma_\gamma) = 1$ for $\gamma \in \Gamma$, and $\phi(\epsilon) = 0$.

Define $\mathbf{trans}_\Gamma^\Sigma$ as follows:

1. $\mathbf{trans}_\Gamma^\Sigma(e) = \epsilon$ if e is the empty string.
2. $\mathbf{trans}_\Gamma^\Sigma(\gamma s) = \sigma_\gamma(\mathbf{trans}_\Gamma^\Sigma(s))$ if $\gamma \in \Gamma$ and $s \in \Gamma^*$.

Define $\mathbf{trans}_\Sigma^\Gamma$ as follows:

1. $\mathbf{trans}_\Sigma^\Gamma(\epsilon) = e$, where e is the empty string.
2. $\mathbf{trans}_\Sigma^\Gamma(\sigma_\gamma(t)) = \gamma\mathbf{trans}_\Sigma^\Gamma(t)$ if $t \in Term_{\Sigma\emptyset}$

Remark 1.2.2. Clearly $\mathbf{trans}_\Gamma^\Sigma$ and $\mathbf{trans}_\Sigma^\Gamma$ are inverses establishing a bijection between Γ^* and $Term_{\Sigma\emptyset}$.

The first algorithm translates strings to terms, which can then be compressed using the methods in later sections. Then, after decoding the compressed term using the pattern dictionary, we can apply the second algorithm to obtain the original string. Consequently, the sequential compression problem is a special case of the more general term compression problem. Also, some sequential compression algorithms might suggest extensions to the general case of terms.

1.3. **Complexity.** In the sequential case, there are n subsequences of length 1, $n-1$ of length 2, and so on, so the total number of subsequences is $\frac{n(n-1)}{2}$. We can represent an encoding of a sequence s_n over Γ by another sequence e_n as follows:

$$e_n = \begin{cases} p & \text{if subsequence } p \text{ is replaced starting at } t_n \\ \emptyset & \text{if } t_n \text{ is replaced by a subsequence starting before } n \\ t_n & \text{if } t_n \text{ is not replaced} \end{cases}$$

(where $\emptyset$ denotes an arbitrary unused symbol.) Since there are at most $O(n^2)$ possible patterns for replacement and at most constant $|\Gamma|$ values for each t_n, the number of possible encodings is $O((n^2 + |\Gamma| + 1)^n) = O(n^{2n})$. (Of course, the $\emptyset$s can be omitted from real encodings to conserve space, but the worst-case number of encodings and dictionaries is still at most exponential).

Consider a term t_h of size n which is similar to a complete binary tree with height h in which the internal nodes are labeled with a binary operator and the leaves with constants. Suppose that the number of possible different (inequivalent) patterns that match this term at any position is given by T_h. If $h = 1$ then there are two such patterns: $\mathbf{y}$ and t_i. Otherwise, if $h \geq 1$, then $t_{h+1} = \alpha(t_h, t'_h)$ for some terms t_h, t'_h with height h. Assume as an induction hypothesis that the number of possible patterns matching each of t_h and t'_h is T_h, and the sets of matching patterns are S and S'. Then the set of patterns that can match t_{h+1} at any position is

$$S \cup S' \cup \{\alpha(p, q) \mid p \in S, q \in S'\}.$$

Certainly both S and S' contain a trivial pattern. Without loss let us assume that if $p \in S$ and $q \in S'$ and $p \equiv q$ then $p = q$, so that only inequivalent patterns are counted. Then the trivial pattern $\mathbf{y}$ is in both S and S'. We cannot assume that any other patterns are in $S \cap S'$, or that any of the patterns in $\{\alpha(p, q) \mid p \in S, q \in S'\}$ are also in S or S', so the total possible number of patterns $T_{h+1} = (2T_h - 1) + {T_h}^2 = (T_h + 1)^2 - 2$.

We can prove that $T_h > 2^{2^{h-1}} + 1$ for all $h \geq 2$: For the base case $h = 2$, we have

$$T_2 = 7 > 5 = 2^{2^1} + 1.$$

For the inductive case, assume $T_h > 2^{2^{h-1}} + 1$ for some $h \geq 2$. Then

$$\begin{aligned} T_{h+1} &= (T_h + 1)^2 - 2 > (2^{2^{h-1}} + 2)^2 - 2 \\ &= (2^{2^{h-1}})^2 + 4(2^{2^{h-1}}) + 2 - 2 \\ &= 2^{2^h} + 2^{2^{h-1}+2} > 2^{2^h} + 1 \end{aligned}$$

because for all $h > 2$, $2^{2^{h-1}+2} > 1$. Thus also $T_{h+1} > 2^{2^h} + 1$, which completes the proof.

Consequently in the worst case the set of possible matching patterns of a complete term of height h could be proportional to 2^{2^h}. Since the size $|t_h|$ of t_h is approximately 2^h, the worst-case number of matching patterns is proportional to $O(2^n)$ where n is the size of the term. On the other hand, since a term is isomorphic to a rooted tree, and term patterns are isomorphic to connected subtrees of that tree, there can be at most 2^n different patterns matching any term because there can be at most 2^n different subsets (connected or otherwise) of vertices of the tree. So at worst the number of patterns can be $O(2^n)$, and no smaller bound is possible because in the case of complete binary terms the mumber of patterns can be proportional to 2^n.

By an argument similar to that for sequential encodings we can show that the number of possible encodings is $O((T_{\log n})^n)$, where n is the size of the term to be encoded. Because T_h is $O(2^{2^h})$, the search space for term compression is therefore $O(2^{n^2})$.

2. Pattern Substitution Trees

Substitution trees [12, 11] have been developed for use in term indexing in automatic theorem provers. These trees can store and look up terms efficiently by taking advantage of shared term structure, and to find terms which are similar (in the sense that they unify with a particular term) quickly. In this section we will introduce a new variation on this structure called *pattern substitution trees*, which can be used to store and be index patterns efficiently.

A substitution tree is a tree with edges labeled by substitutions (over special substitution variables) such that every vertex that is not a leaf or the root has more than one child. In a substitution tree, each leaf corresponds to a term resulting from applying the substitutions on the path from the root to the vertex in order. The restriction on the number of children helps ensure that the tree uses space efficiently, because a non-root vertex with only one child can always be optimized out by concatenating the child-edge and parent-edge's substitutions. Terms can be inserted into, looked up, and removed from substitution trees efficiently, and so they are used for term indexing in automated theorem provers.

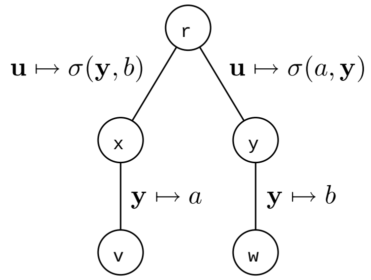

FIGURE 1. Flawed PST

We here introduce a variant of the substitution tree structure, which we will call a *pattern substitution tree (PST)*. Pattern substitution trees are structurally similar to substitution trees, but are used differently: to express and store a set of patterns efficiently by taking advantage of shared pattern structure, rather than to index a set of terms. For that reason, the substitution variables of a pattern substitution tree are the pattern variables of the patterns over a particular operator domain. Both leaves and internal nodes may correspond to patterns that contain pattern variables. We relax the restriction on the number of children a vertex may have in a pattern substitution tree so that patterns that share structure (such as $\sigma(\mathbf{y})$ and $\sigma(\tau(\mathbf{z}))$) can be added to a tree and indexed by different vertices.

2.1. **Definitions.**

Definition 2.1.1. A *pattern substitution tree (PST)* over $Pat^Y_{\Sigma X}$ is a structure $P = (r, \mathbf{u}, V, E, \xi)$ such that $T = (r, V, E)$ is a rooted tree. The function $\xi : E \to Sub^Y_{\Sigma X}$ is called the *substitution (edge-)labeling* of P and $\mathbf{u} \in Y$ is called the *root variable* of P.

The pattern *associated with* a vertex v of a PST is the pattern

$$[\xi(v_{n-1}, v_n)] \cdots [\xi(v_1, v_2)]\mathbf{u},$$

where $(v_1, \dots, v_n)$ is a path from the root to v. The pattern associated with v in P is also denoted $pat_P(v)$.

Remark 2.1.2. If $P = (r, \mathbf{u}, V, E, \xi)$ then we denote $r(P) = r, \mathbf{u}(P) = \mathbf{u}, V(P) = V$, *etc.*

Note that pattern substitution trees as defined above have the potential flaw that more than one vertex might correspond to the same pattern. For example, consider the tree of Figure 1. Vertices v and w both yield the same pattern

$$[\mathbf{y} \mapsto a]\sigma(\mathbf{y}, b) = [\mathbf{y} \mapsto b]\sigma(a, \mathbf{y}) = \sigma(a, b),$$

but obviously occupy different places in the tree. Ordinary substitution trees also exhibit this behavior, but in practice the effort needed to avoid it is unjustifiable (at least, when using substitution trees for term indexing[12].) Here we will also assume that this is not a serious problem.

Definition 2.1.3. A function $\mu : V(P) \to \mathbb{N}$ is called a *multiplicity weighting*.

Remark 2.1.4. A multiplicity weighting is intended to record the number of different places a pattern matches a term. Unfortunately, the difficulty mentioned above complicates matters; for example, if $\mu(v) = 4$ and $\mu(w) = 1$ in the tree of Figure 1, then the multiplicity of $\sigma(a, b)$ is either 4 or 1. In fact, the actual multiplicity of $\sigma(a, b)$ should be 5. In general, though, it may not be practical to seek out all the different vertices which give a pattern and add together their multiplicities to obtain the correct answers.

2.2. **Algorithms.** Substitution trees provide efficient means of inserting and looking up matching terms in an index. Removing terms from the index is not usually as efficient, but term indexing applications do not require efficient removal. Pattern substitution trees are meant to serve different goals, but the algorithms used to meet these goals are similar to the substitution tree algorithms. Here we present descriptions of the algorithms used to insert and look up patterns in a PST, and to find the patterns associated with vertices in PSTs.

In the complexity results claimed, it is assumed that the underlying labeled tree data structure is implemented efficiently with $O(1)$ vertex labeling, insertion, and deletion, for example with hash tables, and that term and substitution representations are not needlessly copied. Substitution matching, generalization, and composition are assumed to have worst case $O(n)$, $O(n^2)$, and $O(n)$ implementations, respectively, with n a bound for the sizes of the substitutions matched, generalized or composed. The sizes of inputs x are denoted $|x|$. The largest substitution in P is denoted $sub\_max(P)$. This value can at worst be $O(|\xi(P)|)$, but in many cases it might be possible to argue that it is $O(1)$ on average.

Algorithm 2.2.1 (**add**).
Input: A PST P, a vertex v, and a substitution s.
Output: A PST that is identical to P except possibly that there is now an edge (v, w) such that $\xi(v, w) = s$. (There might already have been such an edge in P.)
Procedure **add**(P, v, s):
If v already has a child w such that $\xi(v, w) = s$ then simply return P. Otherwise choose $w \notin V(P)$ and return

$$P' = (r, u, V \cup \{w\}, E \cup \{(v, w)\}, \xi \cup \{((v, w), s)\}).$$

Complexity: O(1) time. This is simply insertion into the tree and labeling structure.

Algorithm 2.2.2 (**get_sub**).
Input: Two vertices v and w of P such that v is an ancestor of w.
Output: A substitution s that is the concatenation of all the substitutions on the path from v to w.
Procedure **get_sub**(P, v, w):

1. If $v = w$ then return $\cdot$
2. Otherwise, return $[\xi(p, w)]$**get_sub**(P, v, p), where p is the parent of w.

Complexity: $O(h(P) sub\_max(P))$ time. At most $h(P)$ compositions are needed, where $h(P)$ is the height of P.

Algorithm 2.2.3 (**get_pat**).
Input: A vertex v of P.
Output: The pattern $pat_P(v)$ associated with v in P.
Procedure **get_pat**(P, v):

Return $[\mathbf{get\_sub}(P, r(P), v)]\mathbf{u}(P)$.
Complexity: $O(|P| sub\_max(P))$ time.

Algorithm 2.2.4 (find_matching_sub).
Input: A vertex v of P and a substitution s.
Output: A vertex v' of P and a substitution s' such that v corresponds to the most specific pattern in the tree that matches $[s]pat_P(v)$, and s' is the matching substitution.
Procedure **find_matching_sub**(P, v, s):

1. If there is a child w of v such that $\xi(v, w) \preceq s$ under substitution t then return **find_matching_sub**(P, w, t).
2. Otherwise return (v, s).

Complexity: $O(|P| \max(|s|, sub\_max(P)))$ time.

Algorithm 2.2.5 (lookup).
Input: A pattern p and substitution tree P.
Output: A vertex v' and substitution s' such that v' corresponds to a most specific match in P of p and s' is the matching substitution.
Procedure **lookup**(P, p):
Return **find_matching_sub**$(P, r(P), \mathbf{u}(P) \mapsto p)$.
Complexity: $O(|P| \max(sub\_max(P), |p|))$ time.

Algorithm 2.2.6 (split).
Input: A vertex v of P and a substitution s.
Output: A vertex w and pattern substitution tree P' identical to P except possibly modified so that $[s]pat_P(v) = pat_P(w) \in P$.
Procedure **split**(P, v, s):

1. If there is a child w of v such that $t = \xi(v, w)$ and s have a nontrivial most specific generalization u and that $u_1 \circ u = t$ and $u_2 \circ u = s$. Then choose such a w and
   (a) if $u_1 \equiv u_2 \equiv \cdot$ (that is, $\xi(v, w) \equiv s$) then return (w, P).
   (b) if $u_1 \equiv \cdot$ and $u_2 \not\equiv \cdot$ ($\xi(v, w) \preceq s$) then return **split**(P, w, u_2).
   (c) if $u_1 \not\equiv \cdot$ and $u_2 \equiv \cdot$ ($s \preceq \xi(v, w)$) then choose a vertex $v' \notin V(P)$ and return v' and
   $$\begin{aligned} P' = (&\mathbf{u}(P), r(P), \\ &V(P) \cup \{v'\}, \\ &(E(V) \backslash \{(v, w)\}) \cup \{(v, v'), (v', w)\}, \\ &(\xi \backslash \{((v, w), t)\}) \cup \{((v, v'), u), ((v', w), u_1)\}). \end{aligned}$$
   (d) if $u_1 \not\equiv \cdot$ and $u_2 \not\equiv \cdot$ then choose vertices $v', v'' \notin V(P)$ and return v'' and
   $$\begin{aligned} P' = (&\mathbf{u}(P), r(P), \\ &V(P) \cup \{v', v''\}, \\ &(E(V) \backslash \{(v, w)\}) \cup \{(v, v'), (v', v''), (v', w)\}, \\ &(\xi \backslash \{((v, w), t)\}) \cup \{((v, v'), u), ((v', v''), u_2), ((v', w), u_1)\}). \end{aligned}$$
2. Otherwise there is no child-edge of v labeled with a substitution that shares any structure with s. Return $(v, \mathbf{add}(P, v, s))$.

Complexity: $O(|P|(\max(|s|, sub\_max(P)))^2)$ time.

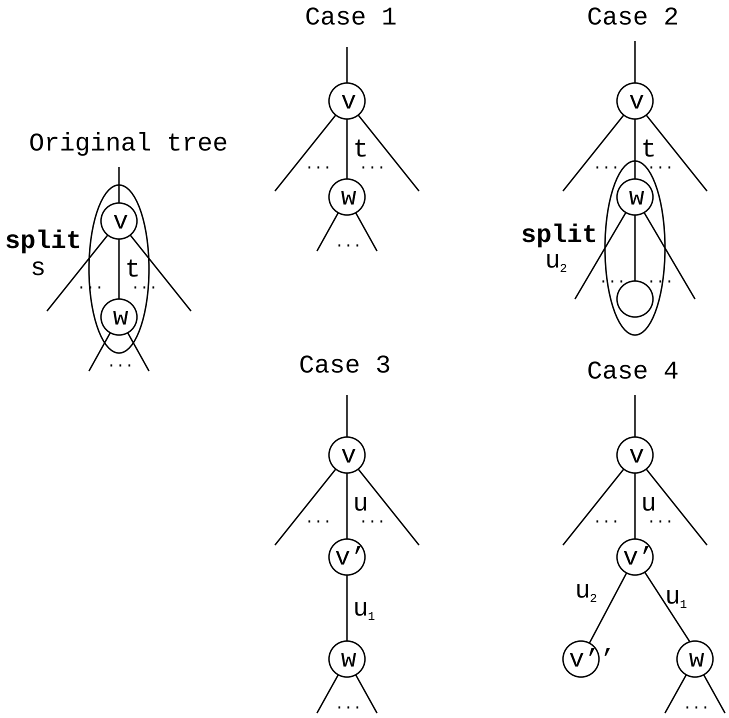

FIGURE 2. `split` cases

Remark 2.2.7. Consider Figure 2. Case 1 illustrates a vertex v with child w and matching substitution $s' = \xi(v, w)$ before the **split** algorithm executes. The other three cases show the action taken in the other three cases of Step 1 above. In each case observe that the patterns associated with v and w cannot have changed. The **split** algorithm adds a substitution to a PST at a vertex in such a way that shared structure (detected by finding generalizations) can be used.

However, if there is ever more than one edge with which a substitution generalizes non-trivially, this algorithm will pick one and ignore the others. This could result in multiple vertices in the substitution tree associated with the same pattern. It could also result in inefficient pattern storage, because if two vertices correspond to the same pattern, and a pattern matching one already present as a child of one of them is **insert**ed, the **split** algorithm might choose the wrong matching vertex and add more duplicate structure to the tree. More seriously for our needs, in doing so the PST might also miss structure shared by the *insert*ed pattern and patterns already in the tree.

Algorithm 2.2.8 (insert).
Input: A pattern substitution tree P and a pattern p.
Output: A vertex v and pattern substitution tree P' identical to P except possibly modified so that $pat_P(v) = p$.

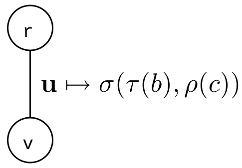

FIGURE 3. Insertion Example (1)

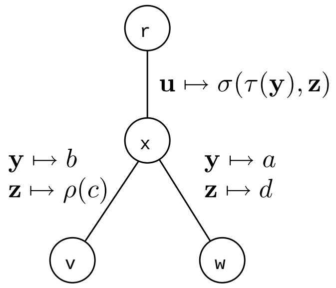

FIGURE 4. Insertion Example (2)

Procedure **insert**(P, p):
Return **split**$(P, \mathbf{u}(P), \mathbf{u}(P) \mapsto p)$.
Complexity: $O(|P|(\max(|p|, sub\_max(P)))^2)$ time.

Remark 2.2.9. Note that if p is already present in P, then **split**$(P, \mathbf{u}(P), \mathbf{u} \mapsto p)$ has the same effect as **lookup**(P, p). If the pattern is already in P then no additional effort is expended in inserting it.

Example 2.2.10. Figure 3 depicts an empty pattern substitution tree to which only the pattern $\sigma(\tau(b), \rho(c))$ has been added. Figure 4 shows the same tree after $\sigma(\tau(a), d)$ has been added to it. Note that after adding the second pattern, the tree was rearranged to take advantage of the structure shared by the two terms, but that vertex v is associated with the same pattern in both trees. This is important because we may wish to use vertices to stand for the patterns associated with them in order to save space.

Remark 2.2.11. We will assume that if there is a multiplicity weighting μ associated with a PST P, then whenever a pattern is inserted into P at v, the value of μ is incremented.

3. SEARCH ALGORITHMS

As discussed above, in the worst case the space complexity of term compression is $O(2^n)$ where n is the size of the set of terms to be compressed. However, in any given instance of the tree compression problem, the set of patterns actually needed in an optimal representation is small (only $O(n)$ space at worst). When we have additional knowledge of the trees we wish to compress, we may be able to restrict our search to small subsets of the whole pattern space in which useful patterns are likely to be found. Our approach will be to calculate small (linear or polynomial space) sets of patterns that are likely to be useful, instead of exhaustively calculating all possible patterns. We will present algorithms that calculate sets of patterns along with their multiplicities in the input terms. The multisets will be represented as pattern substitution trees with multiplicity labelings, in order to try to save space when possible.

In this section we present three algorithm schemes which approach the problem in different ways. The local algorithms work recursively on a term, for each subterm inserting some matching patterns into the set. The global algorithm starts with an initial set of small pattern occurrences, and then iteratively uses the occurrences to build up larger occurrences. The extended Lempel-Ziv algorithm performs an analogue of a Lempel-Ziv scan on the term, using pattern substitution trees to store the observed patterns. For the first two approaches we present a complete algorithm which generates all possible patterns, and approximate algorithms based on the complete algorithm which limit the search space in some way. The approximate algorithms represent compromises which sacrifice some effectiveness for efficiency. The algorithms given work on one term at a time, but the final section discusses extending these algorithms to search for patterns in sets of terms.

3.1. **Local.** The *local* pattern search schemes proceed recursively on the structure of the terms we wish to compress. For each subterm of each term, a local algorithm computes a set of patterns that match that term and adds them to a set of patterns that is carried along the computation. Local algorithms generate patterns that depend only on information "local" to the subterms under consideration; no knowledge of the rest of the term is used.

For the following local algorithm scheme we will assume some efficient structure for storing multisets of patterns is given without necessarily requiring that structure to be a pattern substitution tree. If pattern substitution trees are used, then the union operation can be implemented with successive **insert**s and modifications of μ. We will also assume that some function $f : Term_{\Sigma X} \to \mathcal{P}(Pat^Y_{\Sigma X})$ is given that generates a set of patterns matching a term.

Algorithm 3.1.1 (local).
Input: A term t, multiset $S \subseteq Pat^Y_{\Sigma X}$, and function $f : Term_{\Sigma X} \to Pat^Y_{\Sigma X}$
Output: A multiset of patterns.
Procedure **local(t,f)**:
Return $\mathbf{local'}(t, \emptyset, f)$.
Procedure $\mathbf{local'}(t, S, f)$

1. If t is a constant or variable then return $S \cup f(t)$.
2. If $t = \sigma(t_1, \dots, t_n)$ then let

$$S_1 = \mathbf{local'}(t_1, S, f)$$

and for each i, $2 \leq i \leq n$, let

$$S_i = \mathbf{local'}(t_{i-1}, S_{i-1}, f).$$

Finally, return $S_n \cup f(t)$.

Based on the argument above concerning sets of all possible patterns, we can see how a local algorithm could construct the set of all patterns matching a term. The patterns calculated for each subterm could be all patterns matching the subterm. However, we have already seen that this approach might not be feasible because the number of patterns can to grow exponentially in the worst case. This form of the local algorithm does terminate and generate the complete set of patterns, but it cannot be implemented efficiently. We will refer to this complete local algorithm as (**LC**).

We could limit the size of the set of patterns generated by only generating patterns of height less than a given limit h. Then the number of patterns generated could at most be nT_h, where n is the size of the term and T_h is some (possibly large) constant, as calculated above. Therefore this version of local search is linear space. We will call this height-bounding algorithm ($\mathbf{LC_h}$).

Another approach would be to try to generalize the subterm with all the patterns seen already in the term, adding any new patterns which are seen to the set. This is not strictly a local algorithm, because it uses information from other parts of the term. However, this is essentially what pattern substitution trees do in order to store sets of patterns efficiently, so it is just taking advantage of work that is already being done. If this approach is to be taken then algorithm 3.1.1 can still be used, but the resulting substitution tree must afterwards be interpreted so that not only the patterns specifically inserted into the substitution tree but also the intermediate, "discovered" patterns are seen as members of the set the substitution tree represents. This is the main reason for using pattern substitution trees instead of ordinary substitution trees.

One possibility would be to simply add each subterm to the pattern substitution tree and then consider all the intermediate patterns it generates to be part of the generated set. Suppose the algorithm were applied to a binary term t_h of height h with no shared structure; that is, every operator, constant, and variable in t_h was different. Then each subterm would require a new child of the root of the PST labeled with the substitution $\mathbf{u} \mapsto t$. Since there are about 2^h subterms each of size at most 2^h, the space required is at most $O(2^h \cdot 2^h) = O(n^2)$, where n is the size of the term. Therefore, the space required could be quadratic in the size of the original term.

This PST-based algorithm, which we will call (**LP**), is nearly complete in the sense that it is likely to generate most of the patterns worth knowing about (patterns that occur more than once). Some shared structure might not be noticed by the PST because of the flaw mentioned in Remark 2.2.7. This algorithm is $O(|t|^2)$ space and $O(|t||P|(\max(|t|, sub\_max(P)))^2)$ time in the worst case. Since $|P|$ might require $O(|t|^2)$ space, this is actually $O(|t|^5)$ time, but in practice the time needed for insertion might be constant or linear on average, yielding average $O(|t|)$ or $O(|t|^2)$ time. For large terms **LP** is likely to require too much space to produce useful results on real machines.

Again, we can avoid this difficulty if instead of adding the entire term, we add only a pattern matching part of the term to the substitution tree. For example, we could simply add only the pattern p_h that is the largest pattern that matches the subterm whose height is $\leq h$. Then the space required for the substitution tree would be at most nT_h. This height-bounded PST algorithm ($\mathbf{LP}_h$) is therefore $O(n)$ space, with a possibly large constant factor, and cannot generate patterns of height greater than h.

3.2. **Global.** The *global* algorithm scheme, as its name indicates, searches for useful patterns by considering information from many parts of the term, instead of restricting attention to one subterm at a time. It operates by first generating a pattern occurrence for each position in the term (generally a small one). Then it iterates over all term positions to try to find pairs of occurrences that can be combined to form occurrences of new patterns.

The following algorithm is generalized to any starting set of pattern occurrences and a limiting function f which determines when the algorithm should terminate. Again, it does not require that pattern substitution trees be used to represent sets of patterns.

Algorithm 3.2.1.
Input: A term t and set S of occurrences of t
Output: A set of occurrences of t.
Procedure $\mathbf{global}(t, S)$:

1. Set value `flag` to **false**
2. For each position p of t:
   (a) For each pair of occurrences $c = (l, s), d = (m, t) \in S$ such that $(m, \mathbf{y}) \in s$, if $join_m(c, d)$ is not in S then insert it into S and set `flag` to **true**.
3. If `flag` $=$ **false** then return S, otherwise return $\mathbf{global}(t, S)$.

If the initial set of occurrences is the set of occurrences of linear atomic patterns at each subterm position, then clearly any linear pattern occurrence can be built up from these occurrences. The complete version of the global algorithm generates all the atomic patterns and then, for each position p, join each occurrence starting at p with each occurrence terminating at p and add the resulting occurrence to the set, terminating when an iteration over the entire term yields no new patterns. This form of the algorithm (**GC**) would be complete, but can require exponential space and time to run.

It is not difficult to see how algorithm 3.2.1 could be modified to limit the number of total pattern joins that may occur. Then the running time is at most $O(i|S|^2)$ and the number of patterns calculated $O(i)$. This variation of the algorithm will be called ($\mathbf{GI}_i$). It is not complete, but can be implemented efficiently.

The general algorithm 3.2.1 could also be modified to limit the number of times a pattern is used by using multisets for S and removing one instance of each occurrence whenever it is used to build another occurrence. The number of times an occurrence could be used would be determined by the number of instances it had in the initial multiset of occurrences. We will call this algorithm ($\mathbf{GM}_m$). If each pattern in the multiset had multiplicity at most m, then the running time would be $O(m|S|^3)$ and number of patterns calculated would be $O(m|S|)$.

Also, the algorithm could be implemented more efficiently by indexing the set (or multiset) of occurrences so that neighboring occurrences and occurrences of the same pattern are easy to find. Then many similar pairs of occurrences could be efficiently joined simultaneously. Patterns which only occur once in the term are useless as candidates for compression, so this version of the algorithm could avoid joining pairs of occurrences that are not similar to any other pairs. Consider the following algorithm:

Algorithm 3.2.2 (indexed_global).
Input: A term t and families $(P_p \mid p \in Pat^Y_{\Sigma X})$ (the *pattern* index), $(S_m \mid m \in Pos)$

(the *starting* index) and $(T_l \mid l \in Pos)$ (the *terminating* index) of sets of occurrences of patterns in t
Output: A set of occurrences of t
Procedure **indexed_global**(t, S):

1. Set value `flag` to **false**
2. For each position m of t:
   (a) For each pair $(c, d) \in S_m \times T_m$, suppose $p = pat_t(c)$ and $q = pat_t(d)$. Consider the set $\{(c_1, d_1), \ldots, (c_n, d_n)\}$ of all joinable pairs of occurrences in $P_p \times P_q$, and suppose that for each i, $(1 \leq i \leq n)$, c_i and d_i are joinable at m_i. If $n > 1$ then for each i $(1 \leq i \leq n)$ if the occurrence $(l, s) = join_{m_i}(c_i, d_i)$ is not in S_l or $T_{s(\mathbf{z})}$ for some $\mathbf{z} \in Dom(s)$ then insert it and set `flag` to **true**.
3. If `flag` = **false** then return S, otherwise return **global**(t, S).

This version of the algorithm requires additional time in every iteration to determine how many occurrence pairs are similar to the initial joinable pair of occurrences. At worst this time is $O(\max(|P|)$ where $\max(|P|)$ is the size of the largest member of $(P_p \mid p \in Pat^Y_{\Sigma X})$. The indexed versions of the above algorithms are called (**IGC**). (**IGI**$_i$), and (**IGM**$_m$). Their running times have the same form as those for the unindexed algorithms, except $|P|$ is replaced with $\max(|P|)$. In practice the index sizes are much smaller than the total number of starting occurrences, so this improves running time considerably by avoiding repeatedly trying to combine patterns that cannot be joined.

Once the global algorithm terminates, the multiset of patterns it has generated can be calculated as follows:

Algorithm 3.2.3 (**global_patterns**).
Input: A set S of occurrences of patterns in term t
Output: A set of patterns
Procedure **global_patterns**(t, S):

1. Create a new set P.
2. For each $c \in S$, add $pat(t, c)$ to P.
3. Return P.

Again, intermediate patterns discovered by the PST can also be considered part of the set of patterns generated by global search.

3.3. **Extended Lempel-Ziv.** Rather than adding patterns matching subterms, or building patterns up from smaller occurrences, as in the local and global algorithms respectively, the Extended Lempel-Ziv algorithm performs a fast scan of the tree into a pattern substitution tree. This type of algorithm is inspired by a variation of the Lempel-Ziv compression algorithm, in which the sequential input is scanned to produce a tree which is then used to encode the input. We will briefly present a version of the Lempel-Ziv algorithm for finding patterns in sequences, then give the extended algorithm.

Algorithm 3.3.1 (**lzscan**).
Input: a string of symbols of Γ
Output: a rooted tree $T = (r, \Gamma', E)$ with an edge-labeling $L : E \to \Gamma$.
Procedure **lzscan**(γ):

1. Create a tree $T = (r, \{r\} \cup \{v_\gamma \mid \gamma \in \Gamma\}, \{\{r, v_\gamma\} \mid \gamma \in \Gamma\})$ and labeling L such that $L(\{r, v_\gamma\}) = \gamma$ for each $\gamma \in \Gamma$.
2. Create a variable x and set it to r.
3. Scan the input sequence $\gamma = (\gamma_1, \dots, \gamma_n)$; for each $1 < i < n$ in order perform the following:
   (a) If vertex $x = v$ of T has a child-edge $\{v, v'\}$ labeled with γ_i, then set $x = v'$ and go on to the next symbol.
   (b) If there is no such edge, then create a new vertex $v' \in \Gamma'$ and edge (x, v') labeled with γ_i, set $x = r$, and go on to the next symbol.
4. Return T.

We can see how an analogous algorithm for tree compression could work, by viewing the "tree" constructed in this algorithm as a *substitution* tree, or more specifically, a pattern substitution tree. Because of the sequential structure of the data, no more than one pattern variable would ever be needed in such a PST if it were only used to express patterns of strings, so the simpler approach in which patterns are simply substrings works without requiring explicit pattern variables. When we wish to consider repetitive patterns in trees, however, we need to know more about the locations in the patterns where they can be extended. Thus for term compression we need the additional power and complexity of PSTs.

Here is the Lempel-Ziv-style algorithm for generating patterns. It is parameterized by a function f which maps position substitutions to pattern substitutions. The role of this function is similar to the function parameter in the local algorithm, which chooses the patterns that will be added to the set. In this case, the function chooses a substitution that matches the positional substitution. We will discuss possible definitions for this function shortly.

Algorithm 3.3.2 (elzscan).
Input: A term over operator domain (Σ, ϕ) and variable set X.
Output: A pattern substitution tree P and multiplicity weighting μ.
Procedure $\mathbf{elzscan}(t, f)$:

1. Create an empty PST $P = (r, \mathbf{u}, \{r\}, \emptyset, \emptyset)$.
2. Create a variable x and initialize it to $r(P)$.
3. Create an empty positional substitution s.
4. Create an empty queue Q of positions. Initialize it to only contain $()$.
5. Repeat the following steps until the queue is empty.
   (a) Remove position l from the head of the queue. Set $s = \mathbf{u}(P) \mapsto l$.
   (b) If there is a child-edge leading from x which is labeled with a substitution that matches s under positional substitution s', then set $s = s'$, set x to the child vertex at the other end of the edge, and repeat this step.
   (c) Otherwise, let $u = f(s)$ and let positional substitution s' match u to s. Let $(v, P') = \mathbf{split}(P, x, u)$. Set $P = P'$ and add 1 to the value of $\mu(v)$. For each $(\mathbf{z}, m) \in s'$, insert l into the tail of Q, and set x to $r(P)$.
6. Return P.

The function f could have many different useful definitions, depending on the situation. If $f(s) = \{(\mathbf{z}, s(\mathbf{z})) \mid \mathbf{z} \in Z\}$ then the scanning algorithm would just add the whole term to the PST and then stop. The opposite extreme definition for f would be $f(s) = \{(\mathbf{z}, \mathbf{z}) \mid \mathbf{z} \in Z\}$. In this case, no progress would ever be made and the algorithm would not terminate. The third possibility is to choose for each variable some nontrivial match for each $s(\mathbf{z})$; for example, the linear atomic matching pattern or the $\preceq$-maximal match of height at most h. Choosing the linear atomic matching pattern for each variable in the domain of the substitution seems closest to the original nature of the Lempel-Ziv algorithm.

The Extended Lempel-Ziv algorithm with function f will be denoted $(\mathbf{ELZ}_f)$. The running time depends on f, but if f runs in time proportional to the size of its output then the number of patterns generated is $O(|t|)$. Running time is certainly $O(|t|^2)$, because at most $|t|$ matches requiring $O(|t|)$ time are required. However, the total amount of time required for matching should be $|t|$, because each position in the term is matched at most once. So in fact the time should be $O(|t|)$. The versions of $(\mathbf{ELZ}_f)$ wherein f chooses atomic linear matching substitutions or $\preceq$-maximal matches of height at most h are called $(\mathbf{ELZ}_1)$ and $(\mathbf{ELZ}_h)$ respectively. In either case, f runs in time proportional to the size of its output, so its time and space requirements are the same as those for $(\mathbf{ELZ}_f)$.

3.4. **The general case.** So far in this section all of the algorithms generate patterns and multiplicities present in a single term. It is easy to modify each of these algorithms to search for patterns in a set of several terms $\{t_1, \ldots, t_n\}$.

The local algorithm can be applied to the set $T = \{t_1, \ldots, t_n\}$ of terms as follows:

$$\mathbf{local\_list}(T, f) = \mathbf{local}'(t_n, \mathbf{local}'(t_{n-1}, \ldots \mathbf{local}(t_1, f) \ldots, f), f).$$

The global algorithm could be adapted to sets of terms by introducing new operands σ_i each with rank i, and applying the global algorithm to the term $\sigma_n(t_1, \ldots, t_n)$. If the starting set contains no occurrences beginning at the root (whose patterns contain σ_n), then none of the patterns generated can contain σ_n either.

The Extended Lempel-Ziv algorithm can be modified so that it starts with a given PST instead of creating a new one, and this modified algorithm could be applied to sets of terms, similarly to the local algorithm.

4. Dictionary Algorithms

For all the encoding schemes, the mechanics of constructing and simplifying dictionaries are the same. By separating these algorithms from the encoding algorithms, we can simplify the presentation of the encoding algorithms considerably.

We can construct a dictionary from a set of patterns using the following algorithm:

Algorithm 4.0.1 (make_dictionary).
Input: A set S of patterns.
Output: A dictionary D.
Procedure **make_dictionary**(S):

1. Construct an empty dictionary $D = \emptyset$.
2. For each nontrivial, nonatomic $p \in S$, suppose $Fr(p) = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ and $\sigma' \in \Sigma'$ is an n-ary operator symbol that does not appear in D. Add the rule $(\sigma'(\mathbf{y}_1, \ldots, \mathbf{y}_n), p)$ to D.
3. Return D.

This algorithm does not attempt to take advantage of dependencies among patterns in the set, so any ordering of the resulting rules orders D. The definitions of the rules guarantee that D is reversible. The nontriviality and nonatomicity constraints guarantee that the reverse of D is decreasing with respect to $M_\#$. The other properties are immediate. Therefore, the resulting rewrite system satisfies the definition of a pattern dictionary.

Also observe that dictionaries can contain patterns that are defined in terms of other patterns, as long as the definitions are not circular. This is guaranteed by the fact that dictionaries are orderable. Dictionaries in which dependencies occur can be smaller than dictionaries in which they do not, since the right-hand sides of rules might also contain repetitive structure. The above algorithm does not attempt to take advantage of this possibility. However, if the set of patterns is represented by a PST, then the structure of the PST itself can be used to create dictionaries which use less space because they use prior definitions. However, for such dictionaries the problem of encoding (or finding $\leftarrow_D$-derivations) is more complicated, so for clarity of exposition, we will use only dictionaries in which the rules are independent.

Once terms have been encoded, the dictionary might contain rules that were never directly used. We can remove these unnecessary entries from a dictionary as follows:

Algorithm 4.0.2 (prune).
Input: A list of unused operators and a dictionary
Output: A dictionary identical to D except that all patterns whose left-hand side root operators were not in the list have been removed
Procedure **prune**(o, D):

1. Let $(r_i \mid 1 \leq i \leq n)$ be an ordering of the rules of D chosen according to the definition of orderability.
2. For each i $(1 \leq i \leq m)$, if the root operator of the left-hand side of r_i is not in o, then remove r_i from D and apply r_i to the rest of D until there are no more occurrences of r_i.
3. Return D.

Note that this algorithm might actually increase the size of the dictionary by removing a rule that was not not in the list, but which was used extensively by other rules. If this is the case it might be better to modify the above algorithm so that rules are only removed if doing so would improve the dictionary's cost. For example, if the metric is $N_\#$, rules should only be removed if they occur fewer than twice in the rest of the dictionary. The dictionaries we will use have no definitional dependencies, so we will not worry about this problem further.

5. Encoding Algorithms

Once a pattern multiset has been generated, the input terms must be encoded using the patterns, and a dictionary including definitions for all the operators must be constructed. In this section we will discuss algorithms which attempt to find good encodings of the input set of terms. Finding optimal encodings is probably at least NP-hard, but several approximate approaches might give acceptable, but not always optimal results.

We will explore four approaches. We will first demonstrate that it is possible to solve the problem by brute force, but we will only be able to give an exponential algorithm for doing so. Instead, acceptable solutions may be found by using the *globally best* or *locally best* patterns first. These algorithms proceed by repeatedly making pattern replacements that are guaranteed to improve the measure of the input. Finally, we will suggest a "top-down" algorithm based on the encoding strategy used in Lempel-Ziv compression. In this procedure, the substitution tree is used as a finite automaton in much the same way as the Lempel-Ziv compression tree is used to scan and encode the input.

5.1. **Brute Force.** We have earlier described how all the possible encodings of a term could be enumerated. The brute force algorithm proceeds by generating all the encodings of a list of terms, their associated dictionaries, and their costs under metrics M and N respectively. It returns an encoding which minimizes the sum of the sizes of the terms and the dictionary.

Algorithm 5.1.1 (brute_force).
Input: A term $t \in Term_{\Sigma X}$ and a set S of patterns
Output: An encoded term $u \in Term_{\Sigma \cup \Sigma'} X$ and a dictionary
Procedure **brute_force**(t, S):

1. Construct dictionary $D =$ **make_dictionary**(S)
2. For each u such that $u \leftarrow_D t$, let $D' =$ **prune**$(Op(u) \cap \Sigma', D)$ and calculate $size(u, D') = M(u) + N(D')$.
3. Return the dictionary D' and term u such that $size(u, D')$ is minimized.

Here we make the simplification of assuming some straightforward way of enumerating all possible $\leftarrow_D$-derivations of t is given. D^{-1} is decreasing with respect to $M_{\#}$, so there can be at most $|t| = M_{\#}(t)$ rules used in any derivation of $t' \leftarrow_D t$. There are consequently at most $|D|^{|t|}$ such derivations, if $|D|$ is the number of rules in the dictionary.

This algorithm (**BF**) is correct (it always finds a solution) and optimal (the solution it finds is the best one). It can run in $O(|D|^{|t|})$ time and $O(|t| + |D|)$ space (it needs to remember only the best and the current encoding and dictionary).

5.2. **Local Best First.** The local best-first algorithm attempts to recursively replace matching patterns that maximize the value of a given function. This *heuristic* function $f : Pat^{Y}_{\Sigma X} \times \mathbb{N} \to \mathbb{N}$ estimates the benefits resulting from replacing every occurrence of a pattern p with multiplicity m in the set of terms. We will assume that f is $O(n)$.

Algorithm 5.2.1 (local_best_first).
Input: a term $t \in Term_{\Sigma X}$ and a pattern multiset S
Output: a term $u \in Term_{\Sigma \cup \Sigma' X}$ and dictionary D
Procedure **local_best_first**(t, D, S):
Return **local_best_first**$'(t,$ **make_dictionary**$(S), S)$.
Procedure **local_best_first'**(t, D, S):

1. Let p be a pattern that maximizes $f(p, \mu_S(p))$ among all patterns in S that match t.
2. If p is nontrivial and nonatomic, then
   (a) Let s match p to t and assume $\sigma'(\mathbf{y}_1, \ldots, \mathbf{y}_n) \leftarrow_D p$.
   (b) Return

$$\sigma'(\textbf{local\_best\_first}'([s]\mathbf{y}_1, D, S), \ldots, \textbf{local\_best\_first}'([s]\mathbf{y}_n, D, S).$$

3. If p is trivial or atomic:
   (a) If t is a variable or constant then return t

(b) Otherwise $t = \sigma(t_1, \dots, t_n)$. Return

$$\sigma(\textbf{local\_best\_first}'(t_1, D, S), \dots, \textbf{local\_best\_first}'(t_n, D, S).$$

This algorithm ($\mathbf{LB}_f$) requires $O(|S||t|^3)$ running time: for each position, the best pattern must be found and replaced from a set of $|S|$ of them. Matching requires $|t|$ time and applying the resulting substitution requires $|t|$ time. The results are not always correct or optimal, but may produce acceptable results for appropriate heuristics f.

5.3. **Global Best First.** The global best-first algorithm sorts the patterns in the tree in order of decreasing heuristic value, given some heuristic value function f. Then it replaces as many of the occurrences as it can of the best, second best, *etc.* patterns in decreasing heuristic value.

Algorithm 5.3.1 (global_best_first).
Input: a term $t \in Term_{\Sigma X}$ pattern multiset S
Output: a term $u \in Term_{\Sigma \cup \Sigma' X}$ and dictionary D
Procedure **global_best_first**(t, P):

1. Construct dictionary $D =$ **make_dictionary**(S).
2. If $S = \emptyset$, return t and D.
3. Otherwise, let p be a pattern which maximizes $f(p, \mu_S(p))$ over S.
4. If p is trivial or atomic, go to step 6
5. Otherwise, if $q \leftarrow_D p$, recursively replace every occurrence of p in t with q. (If occurrences overlap, both cannot be replaced, so choose one.)
6. Remove p from S. Go back to step 2.

This algorithm (**GB**) requires at worst $O(|S|^2)$ time to calculate all the heuristic values, $O(|S| \log |S|)$ time to sort the patterns by heuristic value, and could in the worst case take $O(|t||S|)$ time to try to replace each pattern in S at every location in t. Thus the overall time complexity is $O(|S|(|S| + \log |S| + |t|) = O(|S|^2 + |S||t|)$. If the search method generates $O(1)$ or $O(|t|)$ patterns then the required times are $O(|t|)$ or $O(|t^2|)$ respectively. This algorithm is not always correct or optimal, but may produce efficient dictionaries and encodings if the heuristic function is well-chosen.

5.4. **Top-Down.** The following is a version of the Lempel-Ziv encoding algorithm:

Algorithm 5.4.1 (lzencode).
Input: a string of symbols of Γ and a rooted tree $T = (r, \Gamma', E)$ with an edge-labeling $L : E \to \Gamma$.
Output: a string of symbols of $\Gamma \cup \Gamma'$, and function $M : \Gamma' \to E$.
Procedure **lzencode**(γ, T, L):

1. Create variable x initialized to r.
2. Create an empty mapping M.
3. Create an empty string γ'
4. Scan the input sequence $(\gamma_1, \dots, \gamma_n)$; for each $1 < i < n$ in order perform the following:
   (a) If vertex $x = v$ of T has a child-edge $\{v, v'\}$ labeled with γ_i, then set $x = v'$ and go on to the next symbol.
   (b) If there is no such edge and $x = r$ then set $\gamma' = \gamma' \gamma_i$
   (c) If there is no such edge and $M(\sigma') = x$ for some $\sigma' \in \Gamma'$ then set $\gamma' = \gamma' \sigma'$.

   (d) Otherwise find a new symbol $\sigma' \in \Gamma'$ and modify M so that it assigns σ' to x, and set $\gamma' = \gamma'\sigma'$.
5. return γ' and M.

This algorithm can be extended to encode terms using the patterns in a pattern substitution tree, as follows:

Algorithm 5.4.2 (top_down).
Input: a term $t \in Term_{\Sigma X}$ and pattern substitution tree P.
Output: a term $u \in Term_{\Sigma \cup \Sigma' X}$ and dictionary D
Procedure **top_down**(t, P):

1. Create a dictionary $D = \textbf{make\_dictionary}(P)$. (We consider P to represent a pattern set.)
2. If $(v, s) = \textbf{lookup}(P, t)$ and p is nontrivial and nonatomic then
   (a) Let $\sigma'(\mathbf{y}_1, \ldots, \mathbf{y}_n) \leftarrow_D p$. (There must be such a rule in D.)
   (b) Set
$$u = \sigma'(\textbf{elzencode}([s]\mathbf{y}_1, P), \ldots, \textbf{elzencode}([s]\mathbf{y}_n, P)).$$
3. Otherwise, if $t = \sigma(t1, \ldots, t_n)$ then set
$$u = \sigma(\textbf{elzencode}(t_1, P), \ldots, \textbf{elzencode}(t_n, P)).$$
   If $t = c$ or $t = x$ for some constant c or variable x then set $u = t$.
4. Return u and D.

This algorithm (**TD**) finds the term's closest matching pattern and substitution using the substitution tree, encodes that pattern using an operator from the dictionary, and then proceeds recursively on any subterms. If encoding using trivial patterns were allowed, the algorithm would never terminate when run on terms that matched only the trivial pattern. Therefore, when the trivial pattern is the best match, we do not try to encode the term at the root operator, but instead try to encode its subterms.

This algorithm can produce poor results because it will replace large, nonrepetitive patterns when replacing smaller, more recurrent patterns would be better. If, for example, the entire term is present in the pattern substitution tree, then the top-down algorithm will replace it with a single symbol, and add an entry in the dictionary that defines the symbol to be the entire term. For many metrics it is not an improvement to use the dictionary

$$D = u \to t$$

and encoding t by

$$u \leftarrow_D t.$$

The resulting pair of dictionary and encoding requires more space than the original term did, that is, the encoded object and dictionary actually have larger $M_{\#}$ and $N_{\#}$ values than the original term did.

This algorithm is neither optimal nor correct. It can be implemented quickly: it can run in $O(|t||P| \max(sub\_max(P), |t|))$ time, which for a fixed P works out to $O(|t|^2)$ time. In fact, the total amount of matching work required is only $|t|$ so the total time is $O(|t|)$, as in the **ELZ** algorithm.

5.5. **Top-down brute force.** This algorithm (**TB**) is identical to the top-down encoding except that if there is a choice between encoding a term at the root and leaving the root alone and encoding its children, it tries both and returns the smaller encoding.

This encoding is not the same as the brute force encoding because it takes only one term's size into account at a time, and does not take dictionary size into account. Even if it did, it still might miss optimal encodings by making locally good but globally bad choices.

Top-down brute force encoding is again neither optimal nor correct. It runs in $O(2^{|t|}|P| \max(sub\_max(P), |t|))$ time. If $|P|$ is $O(|t|)$ or $O(|t|^2)$ then this is $O(2^|t|)$ time. The top-down brute force algorithm seems more likely to be useful in encoding small terms or parts of terms, because it requires too much time to produce an answer for large terms.

Part 3. Applications and experiments

In this part of the thesis, we will describe an application of the term compression techniques described above to a practical problem, that of compressing proofs in a system for mobile code safety. First we shall give an overview of the relevant parts of that system. We shall then discuss how to use term compression techniques to compress proofs. We present some experimental results, and finally speculate on how term compression techniques could be extended to make it possible to optimize proofs.

1. Proof-Carrying Code

Proof-carrying code (PCC) is a system for mobile code safety [18]. It provides a means for security-conscious *code consumers* to ensure that code they receive from *code producers* over a computer network is safe to execute. In this system, the code producer is responsible for providing a *proof* (in a suitable formal system) that the code adheres to a *safety policy*, made public by the code consumers. When a code consumer wishes to run the code, it can check the proof first to decide whether it is safe. Proof checking is decidable and can be implemented efficiently, while program verification is undecidable and inherently difficult. Thus, the code producer is responsible for doing the *hard* work of proving a program correct, and the code consumer check the resulting proof, which is *easy*. Proof-carrying code frees the code consumer from an unpleasant choice between verifying the program on its own, or blindly trusting it.

Some useful safety policies might include *memory safety*, *termination*, *privacy*, and *correctness*. Memory safety requires that all memory accesses are to legally allocated locations. Termination requires that the program always terminates within a given number of steps. A program provides privacy if it never sends information from a local file to the network without the user's knowledge. A program is correct if it always behaves according to some behavioral specification, which for practical purposes is usually expressed in some formal system. It is well-known that these properties are undecidable (termination, in particular, is precisely the Halting Problem).

A current proof-carrying code system proves properties of programs indirectly, by proving statements in first-order logic which are equivalent to program safety. The logic used by this system is represented in the Edinburgh Logical Framework (LF) [13]. Proofs are represented as proof-typed terms in LF, and proof checking in the PCC logic corresponds to type checking in LF. Because of this correspondence, the proof checker is simply a LF type checker, which only needs to be implemented once.

Proof representations in this system are currently three to four times the size of the code they accompany. For the system to become practical, the proofs need to be made as small as possible, while still being efficiently checkable. One possible approach to optimizing proofs might be to analyze the logic and proof methods used by the theorem prover to develop a set of proof transformations which can be applied to a proof to reduce its complexity. We will take a different approach here, which takes account only of the syntactical structure of proofs, not the semantic structure. As we will see in this section, machine-generated proofs are likely to contain structural redundancy, so we should be able to apply term compression techniques to help optimize them.

1.1. **Verification conditions.** We can abstractly think of a program property as a set P of programs. A program p is said to *satisfy* P if $p \in P$. Program verification or analysis, then, attempts to determine whether a program satisfies any of a variety of program properties.

This problem is notoriously undecidable for many interesting program properties, such as program termination. In fact, if the set of all possible programs is infinite (as is the case in most useful programming languages), there are uncountably many possible program properties, all but countably many of which cannot even be given finite formal descriptions. On the other hand, if a program property P is finite, then the problem of whether a program satisfies it can be formally decided in principle by checking whether it is in the set P.

In between these two extremes are program properties that are finitely describable but contain infinitely many programs. Many of these properties are useful even if they are undecidable in general, because people usually try to write programs that they understand. One way of describing and proving program properties is to transform the problem into one in logic. Some examples of this approach are Hoare's *axiomatic method*[15], *guarded commands* and *weakest preconditions*[6], and *verification conditions*[18]. We will informally describe (first-order) verification conditions in this section.

Definition 1.1.1. A first-order statement VC is a *verification condition* of a program p for a property P in some mathematical model of computation if VC is true in the model if and only if $p \in P$. A *verification condition generator* $VCGen_P$ for a safety policy P is a (typically computable) function such that $VCGen_P(p)$ is a verification condition of p for the property P.

Note that a program may have nontrivially different verification conditions.

Example 1.1.2. For the trivial program property containing all programs, both $\top$ and $\neg(\exists x(\exists y(\exists z(\exists n((n \geq 3) \supset x^n + y^n = z^n)))))$ are verification conditions for any program p (if Fermat's Last Theorem is true; see [25, 8]).

We prefer verification conditions that are easy to prove, so verification condition generators should generate conditions that are as simple as possible.

Example 1.1.3. For illustration, consider a very simple programming language for a random-access register machine with the following instructions:

```
PUT(R,C)      ; put constant value C into R
MOVE(R1, R2)  ; move value in R2 into R1
LOAD(L,R)     ; load value from location L to register R
STORE(L,R)    ; store value from register R to location L
STOP          ; stop the program
```

A program in this language is a sequence of instructions $\mathtt{S}_1$`;` $\dots$ `;` $\mathtt{S}_n$. We will assume that there is a finite set of registers, that memory locations and constants are given by integers, and memory locations do not overlap (*i. e.*, writing to one location never affects what is read at other locations.)

Suppose that we wish to check that programs only access memory in the range from memory address *A* to *B*. In what follows, `I ;` *S* means a single instruction `I` heading a sequence *S* of instructions. A verification condition generator for this very simple safety policy would be:

```
VC_M(PUT(R,C); S)       = VC_M(S)
VC_M(MOVE(R1, R2); S)   = VC_M(S)
VC_M(LOAD(L,R); S)      = (A ≤ L ∧ L ≤ B) ∧ VC_M(S)
VC_M(STORE(L,R); S)     = (A ≤ L ∧ L ≤ B) ∧ VC_M(S)
VC_M(STOP; S)           = ⊤
```

Consider the following two programs:

```
P1:
PUT(R1,0);
MOVE(R2, R1);
STOP

P2:
LOAD(100,R1);
LOAD(101,R2);
STORE(100,R2);
STORE(101,R1);
STOP
```

The first program simply sets two registers to zero, so it should always be memory-safe. The second program exchanges the values at two memory locations, so it is necessary to verify that the memory accesses are legal.

If we calculate the verification conditions of the two programs we get the following results:

$$\begin{aligned}\mathtt{VC}_M(\mathtt{P}_1) &= \top\\ \mathtt{VC}_M(\mathtt{P}_2) &= (A \le 100 \land 100 \le B) \land (A \le 101 \land 101 \le B)\\ &\quad \land (A \le 100 \land 100 \le B) \land (A \le 101 \land 101 \le B) \land \top\end{aligned}$$

As we would wish, the verification condition for the simple program which does not access memory is simply $\top$. For the more complicated program, there is a more complicated condition. Moreover, we can see that there are duplicate goals in the verification condition, since we must verify multiple accesses to the same memory locations. If we needed to produce a compact formal proof of the second condition by hand, we could consider proving $(A \le 100 \land 100 \le B)$ and $(A \le 101 \land 101 \le B)$ as lemmas, and citing the lemmas twice each. Without performing some difficult analysis, an automatic theorem prover will not recognize or tire of this repetition, and will blindly generate identical subproofs for each identical subgoal.

This example presents a misleadingly simple view of the problems of generating verification conditions, but it does illustrate the point that machine-generated verification conditions and their proofs are likely to contain repeated goals or structures of proof rules. In richer programming languages, more complicated verification conditions are needed, so repetitive structure may be more likely. Term compression techniques may be able to use such redundancy to represent proofs more efficiently.

1.2. **The Edinburgh Logical Framework.** A *logical framework* is a formal system in which other logics can be readily represented. Logical frameworks simplify experimentation by implementing the similar aspects of formal systems (such as term structure and substitutions) so that researchers can concentrate on the differences.

The Edinburgh Logical Framework (LF)[13] is one such system. LF is a simply-typed λ-calculus with dependent types. A logic consisting of constants, function symbols, propositions, and proof rules can be represented by a LF signature associating constant symbols with appropriate types. According to the Curry-Howard isomorphism between types and propositions, proof judgements are represented as dependent types and proof derivations are represented as objects of those types. Proof-checking in the represented system corresponds to type-checking in LF.

Many useful formal systems share concepts such as substitution, term structure, proof structure, and proof checking. These facilities can be provided by a fixed implementation of LF, and systems defined using LF can take advantage of them immediately. One such implementation is a logic programming language called Elf[22].

LF is divided into three syntactical classes: *kinds*, *types*, and *objects* (or *terms*). They are defined by the following BNF grammar:

$$
\begin{array}{lll}
\textit{Kinds} & \mathtt{K} ::= & \mathtt{Type} \mid \Pi x{:}\mathtt{A}.\mathtt{K} \\
\textit{Types} & \mathtt{T} ::= & \mathtt{a} \mid \Pi x{:}\mathtt{A}_1.\mathtt{A}_2 \mid \lambda x{:}\mathtt{A}_1.\mathtt{A}_2 \mid \mathtt{A}\ \mathtt{M} \\
\textit{Objects} & \mathtt{M} ::= & \mathtt{c} \mid x \mid \lambda x{:}\mathtt{A}.\mathtt{M} \mid \mathtt{M}_1\ \mathtt{M}_2
\end{array}
$$

The letters a and c represent type constants and object constants respectively, and the letter x denotes any one of a set of variables. The objects and types of the form $\lambda x{:}\mathtt{A}.\mathtt{M}$ or $\lambda x{:}\mathtt{A}.\mathtt{B}$ represent functions (or, for types, *type families*) which can be applied to individual objects of the specified type A. The Π-types represent dependent function types: $\Pi x{:}\mathtt{A}.\mathtt{B}$ denotes the type of functions from objects of type A to type B, which binds x in B, and $\Pi x{:}\mathtt{A}.\mathtt{K}$ denotes the kind populated by type families from objects of type A to types of kind K, which binds x in K. When x does not appear free in B or K, we use the more familiar notation $\mathtt{A} \to \mathtt{B}$ or $\mathtt{A} \to \mathtt{K}$ to represent these types and kinds. Moreover, in an application M N or A M, of an object or a type to another object, the object or type M or A must be a function or a type family, and have the appropriate type or kind.

In addition, variable types and constant types and kinds are bound in LF *contexts* Γ and *signatures* Σ respectively, defined as follows:

$$
\begin{array}{ll}
\Gamma & ::= \cdot \mid \Gamma,\ x{:}\mathtt{A} \\
\Sigma & ::= \cdot \mid \Sigma,\ \mathtt{c{:}A} \mid \Sigma,\ \mathtt{a{:}K}
\end{array}
$$

There are five main judgments in LF:

Σ sig	Σ is a valid signature
$\Sigma \vdash \Gamma$ ctx	Γ is a valid context under Σ
$\Gamma \vdash_\Sigma K$	K is a valid kind
$\Gamma \vdash_\Sigma A : K$	Type A has kind K
$\Gamma \vdash_\Sigma M : A$	Object M has type A

When there is no possibility for confusion we will omit the subscript Σ in the last three judgments. We will not go into the further details of these judgments and rules relating them; these can be found in [13].

Example 1.2.1. The structure of propositions using connectives

$$(\top, \wedge, \supset, \forall)$$

can be represented by the following sequence of type declarations:

```
% basic types
i        : T%peindividuals
```

```
o        : Type % propositions
pf       : o → Type % proof derivations

% predicate functions
true     : o
and      : o → o → o
imp      : o → o → o
all      : (i → o) → o
```

For example, the first-order proposition $\forall x(((P(x) \supset Q(x)) \wedge P(x)) \supset Q(x))$ can be represented in LF by `(all λx:i.(imp (and (imp (P x) (Q x)) (P x)) (Q x)))`. The symbols `P` and `Q` have type `i → o`.

The following LF constant declarations show how the logical structure can be represented in LF:

```
% proof rules
true_i   : pf true
and_i    : ΠA:o.ΠB:o.pf A → pf B→ pf (and A B)
and_el   : ΠA:o.ΠB:o.pf (and A B) → pf A
and_er   : ΠA:o.ΠB:o.pf (and A B) → pf B
imp_i    : ΠA:o.ΠB:o.(pf A → pf B) → pf (imp A B)
imp_e    : ΠA:o.ΠB:o.pf (imp A B) → pf A → pf B
all_i    : ΠA:i → o.Πx:i.pf (A x) → pf (all A)
all_e    : ΠA:i → o.Πx:i.pf (all A) → pf (A x)
```

These declarations correspond to the usual proof rules for the $\wedge$, $\supset$, and $\forall$ connectives. For example, consider the rules for $\wedge$-introduction and $\supset$-introduction:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\mathrm{I} \qquad \frac{\Gamma, u : A \vdash B}{\Gamma \vdash A \supset B} \supset \mathrm{I}^u$$

The LF constant `and_i` takes 4 arguments: two predicates `A` and `B`, and two proofs, proof `p` of `A` and `q` of `B`. The two proof arguments represent the two judgments that are required in the $\wedge$-introduction rule. The other two arguments are parameters that fix precisely which version of the rule is used. The type of a well-formed application of `and_i` to `A`, `B`, `p`, and `q` has type `pf (and A B)`, corresponding to the result of the $\wedge$-introduction rule.

Analogously, the constant `imp_i` takes three arguments: two propositions `A` and `B`, and a function `f` which maps objects of type `pf A` to objects of type `pf B`. This third argument represents the single prerequisite judgment in the $\supset$-introduction rule, and again the first two arguments parameterize the rule. The result of applying the function `imp_i` is an object of type `pf (imp A B)`, which corresponds to the result of the $\supset$-introduction rule. In natural deduction, we can only use the assumption of A in the proof of B; the variable u is used to enforce this scope restriction. In the LF representation of this proof rule, we enforce this restriction by providing a function function `f` from proofs of `A` to proofs of `B`. The body of this function is abstracted over a variable x of type `pf A`, which only has meaning within the function body.

To represent the verification conditions shown above, we could introduce further representations for the constants, function symbols, predicates, and proof rules needed to express properties of natural numbers. We will instead concentrate on aspects of LF that are more relevant to our discussion of proof optimization.

If we represent a logical system L in LF using signature Σ, we can represent proofs in that system as LF terms which we shall refer to as proof terms. An L-proof D of L-sentence p corresponds to a proof term $\ulcorner D \urcorner$ in LF of type $\texttt{pf}\ \ulcorner p \urcorner$, where $\ulcorner\cdot\urcorner$ denotes the LF representations of L-proofs and L-terms. We can also check alleged proof representations by verifying $\cdot \vdash_\Sigma \ulcorner D \urcorner : \texttt{pf}\ulcorner p \urcorner$.

Example 1.2.2. We might have a proof in first-order natural deduction of $(A \wedge B) \supset (B \wedge A)$, as follows:

$$\dfrac{\dfrac{\dfrac{}{u : A \wedge B \vdash A \wedge B}\,u}{u : A \wedge B \vdash B}\wedge\mathrm{E}_R \quad \dfrac{\dfrac{}{u : A \wedge B \vdash A \wedge B}\,u}{u : A \wedge B \vdash A}\wedge\mathrm{E}_L}{\dfrac{u : A \wedge B \vdash B \wedge A}{\vdash (A \wedge B) \supset (B \wedge A)}\supset\mathrm{I}^u}\wedge\mathrm{I}$$

Using the above signature, if $\texttt{A} = \ulcorner A \urcorner$ and $\texttt{B} = \ulcorner B \urcorner$, we can represent $(A \wedge B) \supset (B \wedge A)$ in LF as `(imp (and A B) (and B A))`, and represent the proof rules by their counterpart LF function symbols. So the representation of the proof above is

```
(imp_i (and A B) (and B A)
       (λu:pf (and A B).
       and_i B A (and_er A B u) (and_el A B u)))
```

which has type `pf (imp (and A B) (and B A))`.

Observe that the representation of the proof contains many seemingly redundant arguments. In particular, the predicates `A` and `B` appear six times each in the proof representation. In a proof of a similar theorem where `A` and `B` represent more complicated predicates, these larger predicates will be reproduced six times each. These repeated arguments are needed by the LF type system. However, often they can be inferred from other parts of the proof term or from the type which the term is supposed to have.

There is an implicit representation of LF, called LF_i[19], which allows terms to contain *placeholders* (denoted by `*`) which stand for subterms whose values can be inferred during typechecking. LF_i places some reasonable restrictions on the kinds of terms that it can typecheck. Most immediate to our purposes is the restriction that variables may not occur as *application heads*, that is, that if a term contains a subterm of the form `M N`, the object `M` cannot be a variable.

Example 1.2.3. In a general case in which `A` and `B` are large terms, the above proof can be represented in LF_i without duplicating `A` and `B` unnecessarily:

```
(imp_i * * (λu:pf (and A B).
           and_i * * (and_er * * u) (and_el * * u)))
```

Proofs can be both represented and validated much more efficiently in LF_i.

2. Proof Compression

LF terms are representable as terms over an operator domain, and so term compression techniques can be used to compress proofs. In this section we will show how to translate LF signatures to appropriate operator domains and proof terms to terms over those domains, and discuss how best to use the term compression algorithms to compress the represented proofs.

2.1. **Translating proofs to terms.** There are two different possible methods for translating from LF to an appropriate operator domain. Given an LF signature Σ, we could use an operator domain which defines operators such as λ, Π, and $\bullet$ (application) and represents the constants in the LF signature as constants. Alternately, we might wish to view the LF constants as functions or operators on their own, with appropriate ranks.

In the former approach we can translate more general proofs because in LF, function applications may be partial: function symbols of types that indicate that they can take n arguments might not always be applied to all n arguments. For the same reason we might have more opportunities to generate useful patterns because there might be many occurrences of partial applications. The second method may lead to smaller term representations than the first method. If the proofs we wish to compress do not typically have partial function applications, then extra effort expended trying to deal with such terms and find useful partial patterns would be wasted. Translating proof rules and other LF function symbols as ordinary n-ary function symbols may simplify pattern search.

For simplicity, and because it will result in a faster implementation we will use the second method to represent proof terms. The proofs produced by the PCC system are typically in a normal form that guarantees that there are no partial function applications.

Definition 2.1.1. The *rank* $r(\mathtt{A})$ of type $\mathtt{A}$ is the largest number of arguments to which it can be applied in LF. It can be recursively defined so that if $\mathtt{A} = \Pi \boldsymbol{x}:\mathtt{B}_1.\mathtt{B}_2$ then $r(\mathtt{A}) = r(\mathtt{B}_2) + 1$, otherwise $r(\mathtt{A}) = 0$. The rank $r(\mathtt{K})$ of a kind $\mathtt{K}$ is defined similarly.

The operator domain (Σ_{LF}, ϕ) for the LF signature Σ and variable set X is defined so that

$$\Sigma_{LF} = \{\mathtt{Type}\} \cup \{\Pi x_k, \lambda x_t, \Pi x_t, \lambda x_o \mid x \in X\} \cup \ulcorner\Sigma\urcorner$$

and

$$\ulcorner\cdot\urcorner = \emptyset$$

$$\ulcorner\Sigma', \mathtt{c}:\mathtt{A}\urcorner = \ulcorner\Sigma'\urcorner \cup \{\mathtt{c}\} \qquad \ulcorner\Sigma', \mathtt{a}:\mathtt{K}\urcorner = \ulcorner\Sigma'\urcorner \cup \{\mathtt{a}\}$$

and

$$\phi(\mathtt{a}) = r(K) \text{ if } \Sigma \vdash \mathtt{a}:\mathtt{K} \qquad \phi(\mathtt{c}) = r(A) \text{ if } \Sigma \vdash \mathtt{c}:\mathtt{A}$$

$$\phi(\mathtt{Type}) = 0 \qquad \phi(\sigma) = 2 \text{ otherwise.}$$

Define the translation $\ulcorner\cdot\urcorner$ so that

$$\ulcorner\mathtt{Type}\urcorner = \mathtt{Type} \qquad \ulcorner\Pi\boldsymbol{x}:\mathtt{A}.\mathtt{K}\urcorner = \Pi_k x(\ulcorner\mathtt{A}\urcorner, \ulcorner\mathtt{K}\urcorner)$$

$$\ulcorner\mathtt{a}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n\urcorner = \mathtt{a}(\ulcorner\mathtt{t}_1\urcorner, \ldots, \ulcorner\mathtt{t}_n\urcorner) \qquad \ulcorner\Pi\boldsymbol{x}:\mathtt{A}_1.\mathtt{A}_2\urcorner = \Pi_t x(\ulcorner\mathtt{A}_1\urcorner, \ulcorner\mathtt{A}_2\urcorner)$$

$$\ulcorner\mathtt{c}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n\urcorner = \mathtt{c}(\ulcorner\mathtt{t}_1\urcorner, \ldots, \ulcorner\mathtt{t}_n\urcorner) \qquad \ulcorner\lambda\boldsymbol{x}:\mathtt{A}_1.\mathtt{A}_2\urcorner = \lambda_t x(\ulcorner\mathtt{A}_1\urcorner, \ulcorner\mathtt{A}_2\urcorner)$$

$$\ulcorner\mathtt{x}\urcorner = x \qquad \ulcorner\lambda\boldsymbol{x}:\mathtt{A}.\mathtt{M}\urcorner = \lambda_o x(\ulcorner\mathtt{A}\urcorner, \ulcorner\mathtt{M}\urcorner)$$

The translation of a proof term p, (or indeed any sort of LF term) to a ΣX-term is then $\ulcorner p\urcorner$.

Example 2.1.2. Consider the proof term

$$p = \lambda A:\mathtt{o}.\lambda B:\mathtt{o}.\lambda C:\mathtt{o}.\lambda P:\mathtt{pf}\ (\mathtt{and}\ (\mathtt{and}\ \mathtt{A}\ \mathtt{B})\ \mathtt{C}).$$
$$(\mathtt{and\_el}\ A\ B\ (\mathtt{and\_el}\ (\mathtt{and}\ A\ B)\ \mathtt{C}\ P))$$

Then the translation of p is the term

$$\lambda_t A(\mathtt{o}, \lambda_t B(\mathtt{o}, \lambda_t C(\mathtt{o}, \lambda_t P(\mathtt{pf}(\mathtt{and}(\mathtt{and}(A, B), C)), \\ \mathtt{and\_el}(A, B, \mathtt{and\_el}(\mathtt{and}(A, B), C, P))))))$$

2.2. **Applying term compression.** For small terms, near-exhaustive pattern search (**LP**) and encoding (**TB**) might be necessary to obtain any improvement at all. Approximate algorithms may not be effective for small terms because they are statistically less likely to find useful repetitive patterns when the sample size is small.

For large terms, though, the exhaustive algorithms are too inefficient to be useful. Instead, local algorithms based on pattern height limits ($\mathbf{LP}_h$) or global algorithms with iteration or occurrence limits ($\mathbf{GI}_i$,$\mathbf{GM}_m$) should be feasible, but might require fine-tuning to find the limit levels which optimize the tradeoff between compression performance and efficiency. The local and global search algorithms seem likely to generate small patterns with high multiplicities, so one of the local or global best-first (**LB**,**GB**) or top down (**TD**) encoding algorithms may be the right choice for encoding terms with the results.

The Extended Lempel-Ziv search and encoding algorithms ($\mathbf{ELZ}_1$,**TD**) should also execute quickly, but the results are less predictable. (**ELZ**) seems likely to generate many small patterns with high multiplicities, and few large patterns with low multiplicities. The top-down encoding is likely to perform better when using such pattern sets, so it seems best to use it with the Extended Lempel-Ziv search algorithm. Certainly if the input is a string represented as a term or a "stringlike" term (one with many subterms which are strings) then (**ELZ**) should perform about as well as (**LZ**) does on strings. However, this performance might not scale to terms that are not stringlike.

One heuristic for the encoding algorithms (**LB**) and (**GB**) would be the function $f_1(p, m)$ which calculates the term metric improvement that would result if p could be replaced m times. For example, with the metric $M_{\#}$, the largest possible improvement is $m(S(p)-1)$, where $S(p)$ is the number of operators and term variables in p. A more accurate heuristic $f_2(p, m)$ might also take into account the increase in the dictionary metric caused by including the rule p. Suppose the dictionary metric is $N_{\#}$. The dictionary entry must include a copy of p and one atomic pattern with $|Fr(p)|$ variables and one operator, so the metric increases by $|p|+|Fr(p)|+1$. The value of '$f_2(p, m)$ is then $f_1(p, m) - |p| - |Fr(p)| - 1 = m(S(p)-1) - |p| - |Fr(p)| - 1$.

3. Experimental Results

3.1. **Methodology.** A system supporting term and pattern structure, substitutions, and pattern substitution trees was implemented in Standard ML (using SML/NJ 110.7). Also, ml-yacc and ml-lex were used to implement a parser for PCC proof annotation files, each of which might contain multiple LF proof terms. A set of proof files with LF symbol counts ranging from 3 to 22558 was chosen from those generated by a version of the PCC certifying compiler. These proofs were classified by size, with "small" proofs of up to 750 symbols, "medium" proofs of 750 to 7500 symbols and "large" proofs of more than 7500 symbols. The categories were chosen so that all of the algorithms ran in "real" time and space (that is, quickly enough to produce results within a few minutes) when applied to the small proofs.

Each proof file was parsed into an abstract syntax for LF absent any LF "class" (*i. e.*, object, type, or kind) information. For example, only one Π symbol was used for both Π-types and Π-kinds. Abstract syntax terms were transformed into terms over the operator domain Σ_{LF}, as described in Section 2.1. This operator domain does distinguish between object, type, and kind operators. The translation from "classless" abstract syntax terms to the "classed" operator domain is possible because if an object is valid then the grammar of LF determines the class of each operator in the object. For example, if `c1` $(\lambda x\!:\!(\texttt{c2 0}).(\texttt{c3}\ x))$ is an object then `c1` and `c3` must be object constants, `c2` must be a type (family) constant, and the λ must be an object λ.

The set of terms in each proof file was searched using implementations of local algorithms (**LP**), (**LP**$_3$) (or (**H1**)), (**LP**$_6$) (or (**H2**)), a global algorithm (**IGI**$_1$**M**$_1$) (or (**GL**)) combining indexing, iteration limits, and use limits, and (**ELZ**$_1$) (or (**ELZ**)). The results were encoded using implementations of (**LB**), (**GB**), (**TD**), and (**TB**). The terms were also translated into sequences and compressed using an implementation of the (**LZ**) algorithm, for a rough comparison of term and sequence compression.

The dictionaries were pruned of entries that were never used. Also, possible definitional dependencies were not used to decrease dictionary size, so it might be possible to obtain better results by representing the dictionaries more compactly. The total compression times and sizes of the resulting terms and dictionaries were measured and recorded. Finally, the encodings were decoded and checked against the original proofs to ensure that the encodings were valid.

For brevity, results for only a representative few of the 20 possible combinations are given here. We will give results for each of the search algorithms paired with (**TD**), and for (**ELZ**) paired with each of the encoding algorithms. Only the small and medium proofs were used with (**LP**) or (**IGI**$_1$**M**$_1$), and only the small proofs were used with (**TB**).

3.2. **Experimental data.** Table 1 summarizes the performance of the various algorithm combinations on small, medium, and large proofs. Figures 5 and 6 plot proof size *vs.* time and compressed size for different search techniques, and figures 7 and 8 compare the results for the different encodings. The search summaries also include the results for sequential (**LZ**) compression.

Method	Small	Medium	Large	Overall
(**LPTD**)	0.948	0.771	–	0.787
(**H1TD**)	1.047	0.848	0.631	0.708
(**H2TD**)	0.984	0.822	0.579	0.874
(**GLTD**)	1.268	0.965	–	0.993
(**LZTD**)	0.943	0.755	0.537	0.614
(**ELZTD**)	0.980	0.752	0.524	0.605
(**ELZTB**)	0.983	–	–	0.983
(**ELZLB**)	1.167	0.801	0.630	0.697
(**ELZGB**)	0.917	0.863	0.847	0.854

TABLE 1. Average compression ratio summary

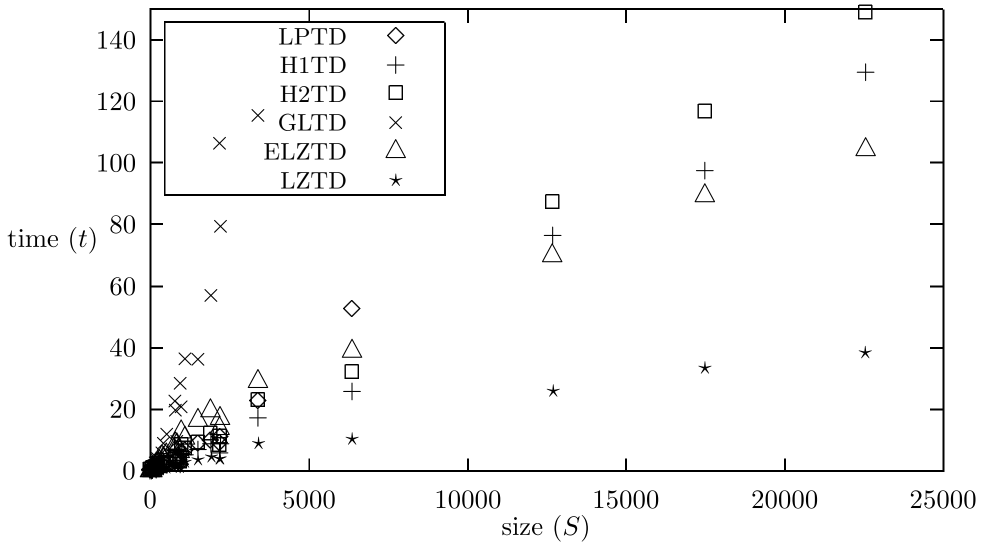

FIGURE 5. Search times

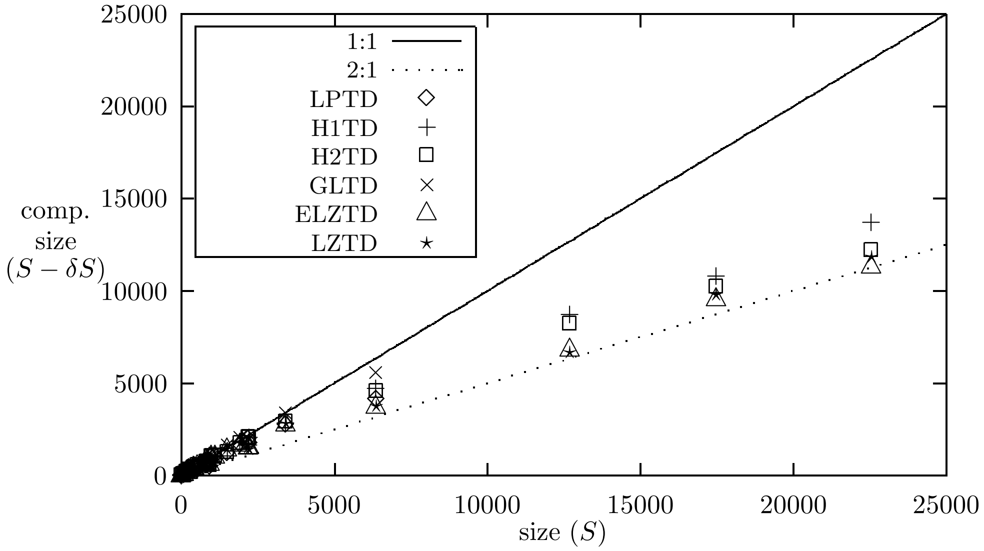

FIGURE 6. Search effectiveness

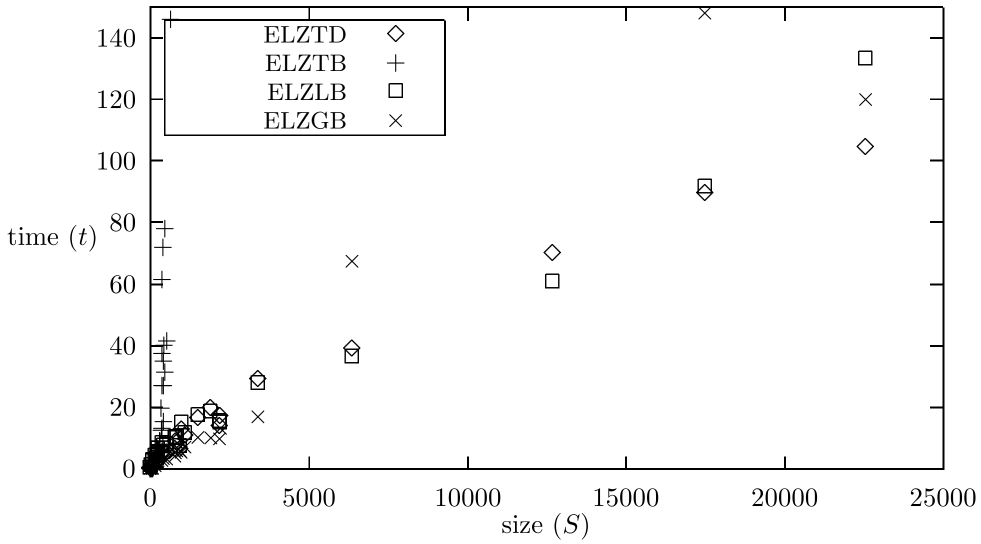

FIGURE 7. Encoding times

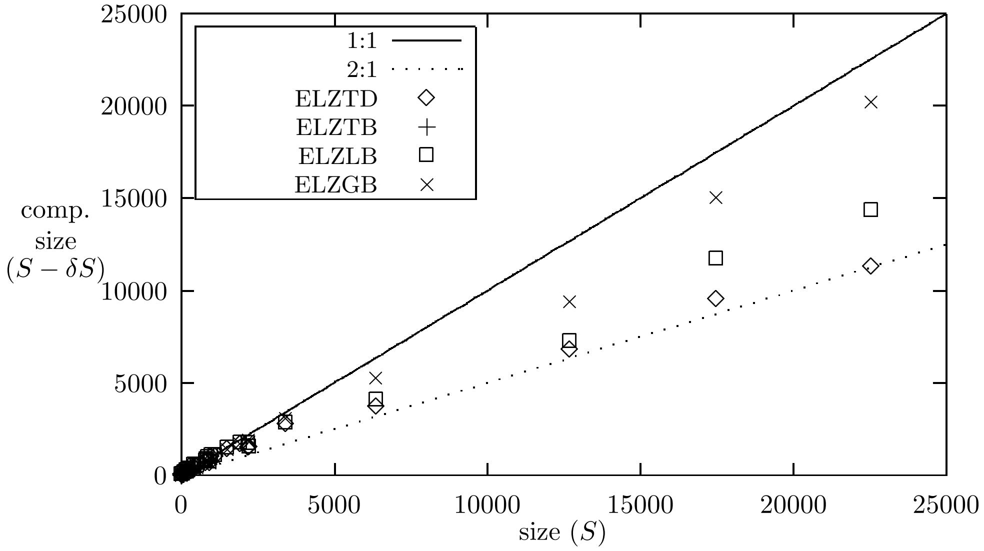

FIGURE 8. Encoding effectiveness

3.3. **Analysis and Interpretation.** The (**LP**) search algorithm appears to run in linear or quadratic time, but requires too much memory to run on the large proofs. The (**GL**) search algorithm appears to require linear or quadratic time also, though if the time is linear then it has a very large constant factor. (**H2**), (**H1**), (**ELZ**), and (**LZ**) also run in linear time, from slowest to fastest in that order. This is probably because the (**H1**) and (**H2**) algorithms perform many insertions into PSTs which might require term generalization, whereas (**ELZ**) only performs term matches, and (**LZ**) only performs string matches.

For small and medium terms, the most effective search seems to be the (**LP**) algorithm, but it cannot run on any larger terms. Algorithms (**ELZ**) and (**H2**) perform about equally well, but both (**H1**) ad (**H2**) perform poorly. For large terms, the (**ELZ**) and (**LZ**) search algorithms have similar effects on size, with the (**ELZ**) algorithm slightly better than (**LZ**) on the two largest terms. Perhaps the fact that (**LZ**) can find incidental sequential patterns that occur in the term linearization makes up for the term patterns it misses. It would be interesting to compare (**LZ**) and (**ELZ**) on larger terms to see which can obtain the best results.

The (**TB**) encoding does require exponential time, so that it cannot run on many terms larger than 1000 symbols quickly enough. The encodings (**GB**) and (**LB**) appear to run in linear or quadratic time on average, with large variations in running time for (**GB**). The (**LB**) algorithm can take longer in the worst case because of the time needed to calculate heuristics, and the (**GB**) algorithm needs additional time to sort the patterns by heuristic. The (**TD**) encoding appears to run in linear time, and with a low constant factor.

The best encoding for small terms seems to be the (**GB**) encoding. The (**TD**) and (**TB**) encoding perform about equally well on small terms, and (**LB**) performs poorly, increasing size by about 17%. For medium and large terms, however, the (**GB**) encoding does not improve and the (**TB**) algorithm is too inefficient to run, while the (**LB**) and (**TD**) algorithms improve substantially.

The most effective term pattern search algorithm when used with the (**TD**) encoding on small terms is the (**LP**) algorithm, and the most effective for medium and large terms is the (**ELZ**) algorithm. The most effective encoding algorithm when used with the (**ELZ**) search is (**TD**). The fastest term search algorithm is also (**ELZ**). The fastest encoding algorithm is (**TD**). In some cases the (**LZ**) search is faster or more effective than any of the others, but on the other hand the patterns generated by the sequential (**LZ**) algorithm are less natural for term compression. None of the omitted combinations of search and encoding techniques performed any better than the combination of (**ELZ**) or (**LP**) and (**TD**).

It is perhaps not very surprising that the Extended Lempel-Ziv algorithm together with the top-down encoding algorithm work well (at times they come close to a 2:1 compression ratio.) These algorithms are based on a near-optimal sequential algorithm, so it would be reasonable to conjecture that the extended algorithms also be near-optimal. The Local Best-First encoding algorithm might suffer because it probably tends to use more different dictionary entries fewer times each, while the Top-Down encoding tends to use only very specific entries. The Global Best-First algorithm may suffer either from replacing too many rare patterns, or not replacing enough common ones, depending on how well its heuristic is balanced.

4. Proof Optimization

Informally, the difference between optimized data and compressed data is that compressed data must be decoded before it can be used, while optimized data can be interpreted as data of the original form From a practical point of view, this difference means that recipients of compressed data must first decompress it, which might require substantial memory and time resources, whereas recipients of optimized data can use the data right away without using additional resources. It could be much harder for code consumers to check compressed proofs than optimized proofs, so in the PCC system, it would be advantageous for code producers to exert additional effort to find optimized proofs in order to make the code consumer's job easier.

So far we have considered the problem of compressing first-order terms; for generality, we have not assumed that any particular rewrite rules apply to them. However, if the terms we wish to compress do have meaning in the context of some rewrite system, as is often the case, it may be possible to express the transformations needed for compression solely as instances of the rules of the rewrite system. Terms transformed in this way could be called *optimized* rather than compressed, because their "encodings" are in fact still *valid terms* which have the same meaning as the original terms in the given semantic context.

For example, in a complete rewrite system, every term has exactly one normal form. Therefore, the relation $\equiv$ defined so that for any terms s, t, $s \equiv t$ if s and t have the same normal form is an equivalence relation, inducing a partition of $Term_{\Sigma X}$. Term optimization could be thought of as an attempt to find for a given term t a smaller (or smallest) term s with $s \equiv t$. Some rules might express "abbreviations" or simplifications that can be always be applied to a term to shorten it. For example, in the arithmetic system, there could be a rule $x + 0 \to x$ which always reduces the size of a term. More general rules like $(\lambda x.s)t \to [x \mapsto t]s$ in untyped or higher-order typed rewrite systems make it possible to express general pattern substitution explicitly in the term, and so pattern search and encoding techniques can be useful to optimize terms in such systems. In general, any partition based on term properties and rewrite rules could be a context for term optimization.

Proofs expressed in LF or LF_i are one example in which optimization might be possible. In this case, the meaning which must be preserved is the type of the proof terms rather than the normal form. A LF proof optimization algorithm could use λ-abstraction to abstract out repeated patterns, reducing proof size. The resulting LF terms could still be checkable as proofs with the same types as the original proof terms, but require less storage space or checking time. We shall see in this section how such optimizations can be made and how term compression might be used to optimize proofs.

4.1. **Proof optimization in LF.**

Pure LF is very expressive. Since proof rules are nothing more than constants with declared types, and proofs are simply terms built up from proof rules, it is perfectly acceptable in LF to use higher-order λ-abstraction to parameterize proofs by proof rules. A proof term P containing several applications of the `and_el` proof rule is β-equivalent to the term $(\lambda \boldsymbol{x}:\Pi A:\mathtt{o}.\Pi B:\mathtt{o}.\mathtt{pf}\ (\mathtt{and}\ A\ B) \to \mathtt{pf}\ A\,.\,P')$ `and_el`, where P' is identical to P except that every occurrence of `and_el` has been replaced by $\boldsymbol{x}$. Since the two terms are β-equivalent, they have the same type, and so the proofs they represent are equivalent.

Moreover, we can λ-abstract over *arbitrary* proof rules, to obtain functions from arbitrary (even invalid) proof rules to proofs. We could never misuse this ability to produce representations of apparent proofs of fallacies, however, because to do so we would need to apply such a function to an object that represented an invalid proof rule. Such an object could only be constructed from already defined, valid proof rules; but any such construction would correspond to a derivation of the new rule in the represented system. Therefore the supposed invalid rule must have been valid. We can therefore provide equivalent proofs of theorems in pure LF by applying λ-abstractions over *derivable* proof rules to *derivations* of those rules.

In general, if a proof contains many repeated patterns of proof rules, we can perform the λ-abstraction procedure outlined above to provide a representation of the proof which is less complex. Each repetition of the pattern is replaced by a single variable symbol, and the entire pattern only needs to appear once, when the parameterized proof is applied to it, so if there are many repetitions of a large pattern, the new equivalent proof will be smaller. When this transformation is performed, we first replace occurrences of patterns in a term with variables, then λ-abstract over $x : \tau$, where τ is the type representing the new proof rule, and finally apply the resulting function to a derivation of the proof rule (that is, a LF term of type τ.)

As in proof compression, it would be useful to have heuristic estimates of the value of a pattern, or the improvement in proof size resulting from using it. To define one such heuristic, we will make a reasonable assumption that variables are assigned a fixed cost (say, 1). If the number of symbols a pattern replaces is S, the size of its type is T, the size of its derivation is D, and the frequency of the pattern in the proof is F, then the improvement in proof measure given by replacing F occurrences of the pattern is

$$F(S-1) - (D+T). \tag{2}$$

We save $S-1$ symbols for each occurrence because we remove S symbols and replace them with one variable symbol, with cost 1. We will always be able to construct the derivation from the pattern by λ-abstracting over the free pattern variables, so we know that $S \leq D$. We know that

$$F(S-1) - (D+T) > 0$$

when

$$F(S-1) > D+T,$$

which is true whenever

$$F(S-1) > (S-1) + T$$

because $D > S-1$. Therefore

$$(F-1)(S-1) > T \tag{3}$$

implies (2)

This formula immediately tells us that replacing a single occurrence of a pattern ($F = 1$) or replacing a pattern with size $S = 1$ can never be beneficial, because the left side of the inequality is zero in either case. This makes sense; replacing patterns does add a fixed amount of complexity to the proof, which can only be offset by savings elsewhere if the pattern replaced is large enough and repeated often enough.

However, it might be the case that a pattern occurs F times in a proof, but fewer than F occurrences of the pattern can actually be replaced at the same time because they overlap. In this case the formula above is too optimistic. It can be corrected by replacing the frequency F with the largest number of occurrences that can be replaced at the same time. Determining this number may be difficult, so in general we will use the frequency instead to (over)estimate the utility of a pattern.

Example 4.1.1. Consider the pattern

$$P = \texttt{(and\_el } \mathbf{A}\ \mathbf{B}\ \texttt{(and\_el (and } \mathbf{A}\ \mathbf{B}\texttt{)}\ \mathbf{C}\ \mathbf{X}\texttt{))},$$

where $\mathbf{A}, \ldots, \mathbf{X}$ are pattern variables which may have many different values. The term

```
λx:pf (and (and (and P Q) R) S).
      (and_el P Q
            (and_el (and P Q) R
                  (and_el (and (and P Q) R) S x))))
```

contains two occurrences of P, but only one of them can be replaced at any one time because they overlap.

Example 4.1.2. Suppose proof p contains seven independent (non-overlapping) instances of P as above. Suppose we λ-abstract over variable

```
andell : ΠA:o.ΠB:o.ΠC:o.
         pf (and (and A B) C) → pf A,
```

replace every occurrence of P with $\texttt{(andell } \mathbf{A}\ \mathbf{B}\ \mathbf{C}\ \mathbf{X}\texttt{)}$, and apply the resulting function to the derived rule

```
λA:o.λB:o.λC:o.
      λX:pf (and (and A B) C).
      (and_el A B (and_el (and A B) C X)).
```

We obtain an object which is equivalent to our original proof. As an estimate of size, we will count the symbols appearing in the type, pattern, and derivation, obtaining $T = 18$, $D = 26$, $S = 3$. So according to our formula the improvement from making this replacement should be $7(3-1) - (18+26) = 14 - 44 = -30$ symbols; that is, the size should *increase* by 30. In general, the savings for replacing F independent occurrences of `andell` is $2F - 44$. Abstracting a proof rule introduces a one-time cost, independent of the number of proof rules replaced. In this case, the benefits of replacement outweigh this cost only when $F > 22$, because the cost is large compared to the benefit of replacing only a few instances.

4.2. **Proof compression and optimization.** As we have seen, terms in both LF and LF_i can be optimized by replacing occurrences of patterns with variables or constants that have been given appropriate types and values. In earlier sections we saw how term compression algorithms could be used to compress proof terms. To perform proof optimizations we could use many of the same techniques.

The pattern search techniques could be used unchanged to detect patterns. However, some of the patterns discovered by the pattern search algorithms might not be legal patterns for proof optimization. Using some patterns might require λ-abstracting over types, which is not allowed in LF. Patterns might also separate a variable occurrence from its binder or violate the LF_i restriction on application heads. Another complication is that types must be found for patterns which are to be used as optimizations. Such types must be general enough to be valid everywhere the patterns are to be replaced.

The encoding techniques also require changes to yield appropriate transformation techniques for optimization. In λ-abstracting over proof rules, variables with appropriate types are introduced and then used with appropriate arguments. There are then two stages to transformation: rewriting occurrences of patterns, and adding the necessary λ-abstractions and applications to the root of the term. The rewrite rules used for occurrences are no longer of the form $\sigma(\mathbf{y}_1, \ldots, \mathbf{y}_n) \leftarrow p$ but of the form $(\boldsymbol{x}\ \ \mathbf{y}_1 \cdots \mathbf{y}_n) \leftarrow p$, where $\boldsymbol{x}$ is an LF variable. Then for each $\boldsymbol{x}$ used in rewriting the term, a redex $(\lambda \boldsymbol{x} : \tau . \cdot)\ \ p$ must enclose the term. These steps parallel the encoding and dictionary pruning phases of term compression.

Therefore, the compression techniques must be modified before they can be used to optimize LF terms. Each of the problems described above seem possible to solve. Patterns which cannot be used as optimizations could be generalized to patterns that can, or they could simply not be used. Also, using higher-order generalization[21], we could avoid discovering patterns that separate binders from bound variables. The problem of finding types for patterns with variables of unknown type seems similar to type reconstruction for LF_i terms, which is solved in [19]. Finally, a more general scheme for optimizing higher-order terms or terms with respect to arbitrary rewrite systems could be adapted to the particular problem of proof optimization.

Concluding remarks

1. Related and Further Work

Term rewriting, data compression, and substitution tree indexing are all closely related to term compression. There is some similarity between substitution trees and *tree automata* (described in [10]). A study of tree structure pattern enumeration can be found in [5]. Term pattern search is similar in some respects to common subexpression elimination [1]. If terms contain many repeated closed subterms (that is, subexpressions) and few repeated open patterns, then it would be more efficient to simply use CSE without bothering to look for general patterns. Also, perhaps CSE methods could be modified for use in pattern search. Machine code can have shared structure that can be discovered by program analysis; an approach to compressing machine code for transmission and execution is found in [7]. An approach to compressing syntax trees for transmission and run-time optimization is given in [9].

Further work in the area of term compression might involve finding more effective or efficient techniques for first-order terms, perhaps by extending other established compression algorithms to operate on terms, combining term compression with other approaches, or fine-tuning some of the approaches given here. It would also be interesting to study the effects of term compression techniques to other kinds of term-structured data. Another area for further study might be extending first-order techniques to higher-order or typed terms, or terms with respect to more general rewriting systems. Finally, research could be done in the area of term *optimization* as envisioned in Section 4, either for specific systems such as LF, or for arbitrary typed, higher-order, or general rewrite or semantic systems.

Many structures of interest both to computer scientists and mathematicians can be expressed using first-order terms. Algebraic and symbolic expressions, logical proofs, functional programs and abstract syntax trees are all examples of data with term structure. General, effective term compression techniques would provide means for storing and transmitting such objects efficiently. Perhaps more importantly, redundant structure discovered by pattern search techniques might identify possible code optimizations in syntax trees and functional programs, interesting lemmas and theorems in logical proofs, and interesting functions or constants in symbolic terms.

Until now, the ability and responsibility for noticing such patterns has rested solely with humans, who then might automate the process of detecting fixed patterns and performing appropriate actions such as program optimizations. Now, some assistance is available in the form of pattern search techniques. These techniques could be used to help investigators discover interesting examples of redundancy in data which humans would be unable or uninclined to analyze. Sufficiently advanced search techniques might lead to automatic optimized compiler generation, improved computer-assisted mathematics, or improved automatic theorem proving techniques. Of course, much more research in this and many other areas would be required before such eventualities could come to pass.

2. Conclusions

Several term compression algorithms have been presented, and an implementation of these techniques has proven to be effective in compressing terms representing proofs in a system for program verification. Experimental results show that term compression can be at least as effective as sequential compression, with the added advantage that term compression expressed term structural redundancy more naturally. For that reason, it should be possible to extend term compression methods to find optimized proofs rather than compressed proofs; this would not always be possible if the terms were sequentially compressed. Optimized proofs would be checkable without requiring decoding, and so would be an improvement over compressed proofs in some situations.

Term compression is a promising technique for compressing first-order terms. Before the value of term compression can be established, it is necessary to show that it can successfully compress many different kinds of terms. Also, the techniques given here are not very efficient, and further study is needed to find improved versions or other, better approaches.

3. Acknowledgements

The author wishes to thank Peter Lee, George Necula, Frank Pfenning, and Peter Andrews for their helpful comments and advice in conducting this research and writing this thesis.

References

[1] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[2] Arthur L. Delcher and Simon Kasif. Efficient parallel term matching and anti-unification. *Journal of Automated Reasoning*, 9(3):391–406, December 1992.

[3] N. Dershowitz and J. P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, Amsterdam, 1990. North-Holland. To appear.

[4] Nachum Dershowitz. A taste of rewriting. Technical report 93-5, Leibnitz Center for Research in Computer Science, Hebrew University, Jerusalem, Israel, 1993.

[5] Nachum Dershowitz and Shmuel Zaks. Patterns in trees. *Ninth Colloquium on Trees in Algebra and Programming*, pages 93–102, March 1984.

[6] E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *CACM*, 18(8):453–457, August 1975.

[7] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 358–365, New York, June15–18 1997. ACM Press.

[8] Faltings. The proof of fermat's last theorem by R. taylor and A. wiles. *NOTICES: Notices of the American Mathematical Society*, 42, 1995.

[9] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. *Lecture Notes in Computer Science*, 1222:263–??, 1997.

[10] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[11] Peter Graf. Substitution tree indexing. In Jieh Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *LNCS*, pages 117–131, Kaiserslautern, Germany, April 5–7 1995. Springer-Verlag.

[12] Peter Graf. *Term indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence and Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1996.

[13] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 194–204. IEEE, June 1987.

[14] G. Held and T. R. Marshall. *Data Compression: Techniques and Applications: Hardware and Software Considerations.* Chichester, UK, 1991.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 26(1):53–56, January 1983.

[16] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[17] G. M. Kuper, K. W. McAloon, K. V. Palem, and K. J. Perry. An efficient algorithm for anti-unification. Technical Report RC 14065, IBM Yorktown Heights, February 1988.

[18] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 January 1997.

[19] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. Technical Report CS-TR 97-172, Carnegie Mellon University, 1997.

[20] M. S. Paterson and M. N. Wegman. Linear unification. In *Conference Record of the Eighth Annual ACM Symposium on Theory of Computing*, pages 181–186, Hershey, Pennsylvania, 3–5 May 1976.

[21] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[22] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

[23] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, January 1973.

[24] Richard Taylor and Andrew Wiles. Ring-theoretic properties of certain Hecke algebras. *Annals of Mathematics*, 142:553–572, 1995.

[25] Andrew Wiles. Modular elliptic curves and Fermat's Last Theorem. *Annals of Mathematics*, 142:443–551, 1995.

[26] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23, MAY1977 337-343.