Incremental Computation via Function Caching

William Pugh[†] *and Tim Teitelbaum*[††]
Dept. of Computer Science, Cornell University, Ithaca, NY 14853

1 Introduction

Incremental computation is the technique of efficiently updating the result of a computation when the input is changed. This idea is used in doing semantic checking in programming environments, document formatting in WYSIWYG editors and many other applications. From a different perspective, incremental computation concerns the efficient on-line computation of $f(x_0), f(x_1), \ldots$ where f is some program and $x_0, x_1, \ldots$ is a sequence of input values, each differing from its predecessor only slightly.

It is possible to achieve incremental computation by writing an explicitly incremental algorithm — an algorithm that not only specifies how to compute the output from the input, but also specifies how to update the output when the input changes. This can be difficult and error-prone, so when possible we would prefer to use an incremental evaluator — an evaluator that uses just a description of how to compute the output from the input and is responsible for determining how to update the output correctly and efficiently when the input changes.

Incremental attribute grammar evaluation and incremental dependency graph evaluation have proven to be useful and efficient paradigms for producing incremental evaluators [DRT81] [Rep82] [Rep84] [Hoo86] [HT86] [ACR+87] [Hoo87] [YS88]. Unfortunately, they are only suitable for certain kinds of problems. For example, while it it possible to use such techniques for incremental proof verification [RA84], it seems impossible to use these techniques for incremental theorem proving. We present a new paradigm for incremental evaluation based on function caching that works well in some situations for which incremental attribute grammar evaluation and incremental dependency graph evaluation techniques are unusable. Our paradigm also works reasonably well in many of those situations best suited for incremental attribute grammar evaluation and incremental dependency graph evaluation. We give a comparison of our techniques and Reps's optimal incremental attribute grammar algorithm in Section 7.

[†] Supported by an AT&T Bell Labs Scholarship. Current address: Dept. of Comp. Sci., Univ. of Maryland, College Park, Md, 20742
[††] Supported by NSF and ONR grant CCR85-14862.

2 An Incremental Evaluator for Functional Languages

In its most elementary form, incremental computation is the technique of saving time by reusing or updating the results of previous computations. Function caching, or memoising [Mic68], is the technique of remembering the results of previous function calls and saving time by avoiding their recomputation (when the result of a function call is computed, the call and the result are stored in a function cache; a later function call need not be performed if the result is already stored in the cache). Considering the similarity of our goal with the technique of function caching, it seems promising to examine the usefulness of function caching for incremental evaluation.

Function caching has typically been advocated for uses other than incremental evaluation. Many recursive functions have a pattern of repetitively requesting the value of certain function calls. The classical example of a function whose evaluation can be improved by function caching is the recursive definition of the Fibonacci function, given below. Function caching reduces the time requirements for this function from exponential to linear.

$$fib(n) \equiv \textbf{if } n < 2 \textbf{ then } 1 \textbf{ else } fib(n-1) + fib(n-2) \textbf{ fi}$$

2.1 Using Function Caching to Obtain Incremental Evaluation

The best way to get a quick understanding of situations in which function caching gives us incremental computation is to look at some examples. The function *sl* (for *square_list*) in Figure 2.1 yields a list of the squares of a list of numbers. Figure 2.2 shows the results of a sample use of this program with function caching. The center column shows the function calls produced by an initial request for the value of *sl*[1, 2, 3, 4, 5, 6, 7, 8]. After the computation is complete, the left column shows the results of making a request for the value of *sl*[0, 2, 3, ..., 8] – reflecting a change in the first element. The request for *sl*[0, 2, 3, ..., 8] invokes a request for the value of *sl*[2, ..., 8], which is in the cache. Here, function caching provides incremental performance. The right column shows what would happen if, instead, the last element of the list were changed to 9, and a request for the value of *sl*[1, ..., 6, 7, 9] were made. No significant results from the computation of *sl*[1, 2, 3, 4, 5, 6, 7, 8] could be used.

```
sl(L) ≡      { square list }
  If null(L) then L
    else cons(head(L) * head(L), sl(tail(L)))
```

FIGURE 2.1 - Algorithm to produce the squares of the numbers in a list

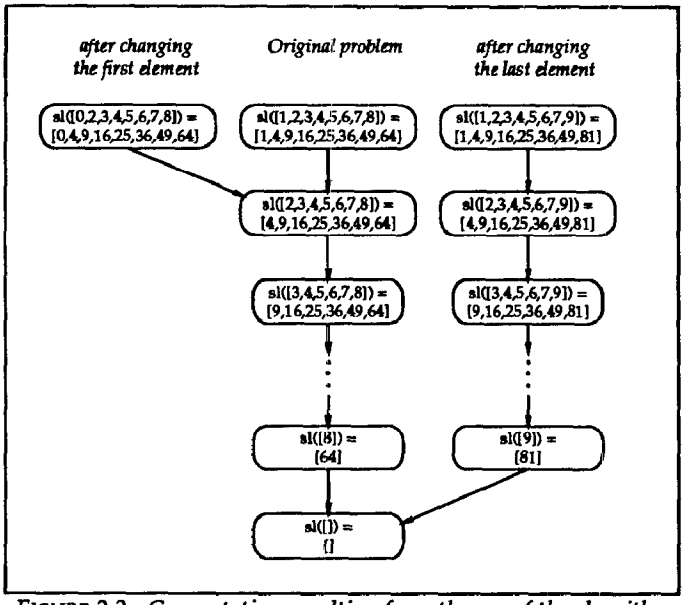

FIGURE 2.2 - Computation resulting from the use of the algorithm in Figure 2.1

Assume that we represent sequences in a way that allows us to efficiently divide them into two nearly equal subsequences and that we rewrite the algorithm in a divide and conquer style, as in the function *ss* (for *square_sequence*) shown in Figure 2.3. This produces the results shown in Figure 2.4 and Figure 2.5. If function caching imposes only a constant factor overhead and first_half, second_half, and append are constant-time functions, only $O(\log n)$ time will be required to compute the new answer. Note that this example is intended only as an illustrative example; it is simple enough that we would probably use an explicitly incremental algorithm.

What happens if an element is inserted into or deleted from the sequence? Assume the decomposition operators first_half and second_half split the sequence exactly in half if the sequence has an even number of elements and if the sequence has an odd number of elements, the extra element goes into the second half. When an element is inserted at the front of the sequence, this produces the results shown in Figure 2.6; almost the entire computation needs to be redone.

```
ss(S) ≡      { square sequence }
  If length(S) = 1 then list(head(S) * head(S))
    else append(ss(first_half(S)), ss(second_half(S)))
```

FIGURE 2.3 - Divide and conquer version of Figure 2.1.

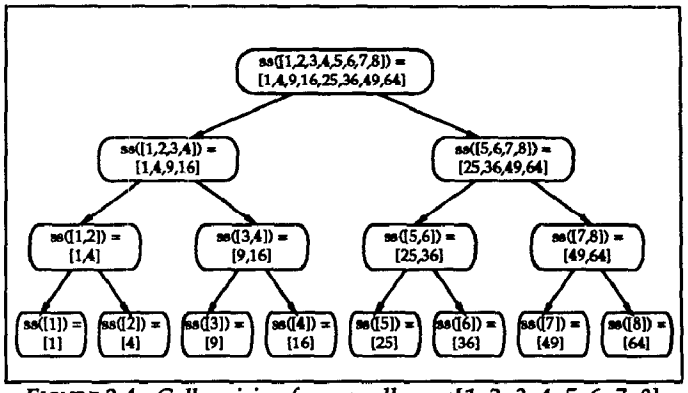

FIGURE 2.4 - Calls arising from a call on ss[1, 2, 3, 4, 5, 6, 7, 8].

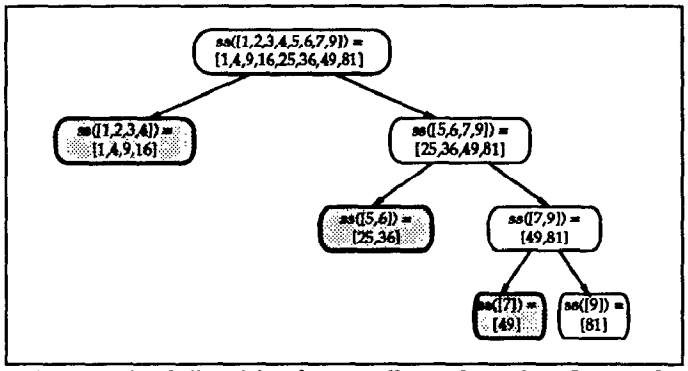

FIGURE 2.5 - Calls arising from a call on ss[1, 2, 3, 4, 5, 6, 7, 9]. *Entries found in the cache are shown in grey.*

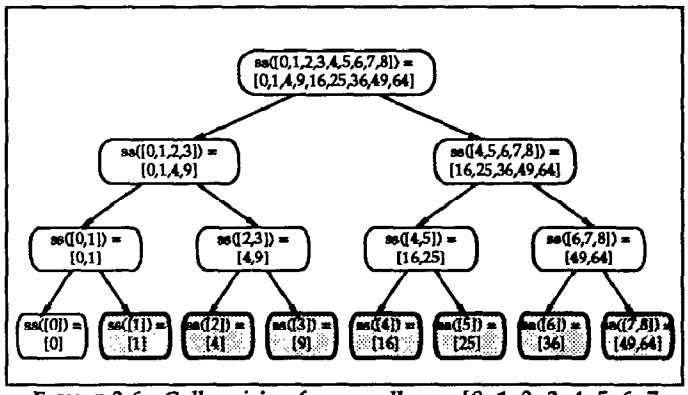

FIGURE 2.6 - Calls arising from a call on ss[0, 1, 2, 3, 4, 5, 6, 7, 8]. *Entries found in the cache are shown in grey.*

We might represent a sequence as a balanced binary tree and use the obvious decomposition rule that splits the sequence represented by a tree T into the sequences represented by the left and right subtrees of T. If T represents the sequence [1, 2, ..., 8], and T' is the tree returned by a function that inserted an 0 at the front of the sequence represented by T, then computations involving T and T' would share common subproblems. However, if T represented the sequence [1, 2, ..., 8], and T' represented the sequence [0, 1, 2, ..., 8], but T' were not derived from T, the decompositions of T and T' might not be similar, and computations involving T and T' might not share any common subproblems.

To exploit function caching, two similar problems must be broken down into sub-problems such that they share many common sub-problems. This explains much of the behavior

Definition 3.3 (O_{prob}). Let $X_{(n)}$ be a random variable with a probability distribution that is dependent on n (e.g., $X_{(n)} =_{prob} NB(n, p)$). We define $X_{(n)}$ to be $O_{prob}(f(n))$ iff

$$\forall \varepsilon \text{ s.t. } \varepsilon > 0, \exists M, n_0 \text{ s.t. } \forall n \text{ s.t. } n \geq n_0, \quad \text{Prob}\{ X_{(n)} \leq M f(n) \} \leq \varepsilon. \square$$

We introduce Theorem 3.1 to allow us to show probabilistic order bounds on algorithms.

Theorem 3.1. Let $X_{(n)}$ be a random variable with a probability distribution that is dependent on n.

$$X_{(n)} \text{ is } O_{prob}(\text{avg}(X_{(n)}) + \text{stndrd\_dev}(X_{(n)})).$$

Proof. By Chebyshev's inequality,

$$\text{Prob}\{ X_{(n)} \geq \text{avg}(X_{(n)}) + M \text{ stndrd\_dev}(X_{(n)}) \} \leq 1/M^2.$$

Given ε, $\varepsilon > 0$, let $M = 1/\sqrt{\varepsilon}$. $\square$

Theorem 3.2. If $X_{(n)}$ and $Y_{(n)}$ are random variables with probability distributions that are dependent on n,

$$X_{(n)} \leq_{prob} Y_{(n)} \text{ and } Y_{(n)} \text{ is } O_{prob}(f(n)) \Rightarrow X_{(n)} \text{ is } O_{prob}(f(n)).$$

Proof. Follows directly from definition of partial ordering on random variables and the definition of O_{prob}. $\square$

4 Data Structures and Algorithms for Incremental Computation

Functional programming languages (and function caching implementations) typically support only simple types such as S-expressions or tagged-tuples[1] as values. Many algorithms are designed to work with more complicated types such as sets or sequences. Rather than extend the interpreter and the function caching system to handle other data types, values in other type systems are represented using S-expressions or tagged-tuples. This sections describes, in general, how problems and values can be represented in a way such that function caching provides efficient incremental evaluation. Sections 5 and 6 present specific solutions for sequences and sets motivated by the discussion in this section.

[1] In sections 5 and 6, we describe data structures using tagged tuples: the value Id$\langle x_0, x_1, ..., x_{k-1} \rangle$ is the tuple tagged with the token Id and containing the values $x_0, x_1, ..., x_{k-1}$. Each tag is associated with a fixed arity.

Tagged tuples are used for clarity of expression. The tagged tuple Id$\langle x_0, x_1, ..., x_{k-1} \rangle$ can be thought of syntactic sugar for the S-expression $\langle$ Id, $\langle x_0, \langle x_1, ... \langle x_{k-1}$, NIL $\rangle ... \rangle\rangle$. Tagged tuples can be combined with a type system to produce typed-tuples; typed-tuples are supported by many functional languages (e.g., ML).

4.1 Representation schemes

A representation scheme is used to represent values of an abstract type τ as values of a concrete type σ. Typically, the abstract type τ is a type that we would like to use in an algorithm but is not a base type of the system in use. For example, if lists are supported by the underlying system but sets are not, we can use a representation scheme in which a set is represented by a list of the elements of the set. In this representation, the concrete list values [6, 3, 'x'] and ['x', 6, 3] would both represent the abstract set value {3, 6, 'x'}.

Definition 4.1 (*representation function*). We can formalize a representation scheme by specifying a representation function that, when applied to a concrete value, returns the abstract value represented by that concrete value. $\square$

Example: If R is a representation function that maps lists to sets, $R([6, 3, 'x']) = R([3, 'x', 6]) = \{3, 6, 'x'\}$.

Note that a representation function is not implemented, but is simply used for arguing about the correctness of algorithms designed to work with that representation. If $R_1 : \rho \rightarrow \sigma$ and $R_2 : \sigma \rightarrow \tau$ are representation functions, $R_2 \circ R_1$ is a representation function giving the value in τ represented by a value in ρ.

Typically, the representation we choose for an abstract type depends on the operations we need to perform and the efficiency with which those operations can be supported by the representation. For incremental computation, we also have to take into consideration the fact that we intend to use function caching and that we wish to obtain efficient incremental evaluation. The effects of these considerations are discussed in Sections 4.2 and 4.3.

4.2 Data structures and algorithms for function caching

Data structures that we plan to use with function caching must be updatable applicatively and should provide unique representations.

Applicative updates. The use of function caching precludes the destructive updating of data structures. Updates must produce new data structures representing the desired values instead of overwriting existing data structures.

Unique representations. In order to make function caching happen at the abstract level, we must use representations schemes with *unique representations*. A representation function R provides unique representations iff R is a one-to-one function. Note that if R is a one-to-one function, R^{-1} is well-defined. The following example explains our motivation. Let f be a function on sets that is designed to use a representation scheme in which sets are represented as lists and let R be the representation function that maps lists to sets. If x and y are concrete values representing the same set (i.e., $R(x) = R(y)$), we want to be able to reuse a

seen above; divide and conquer algorithms should be used, because they solve a problem by breaking it down into sub-problems, allowing for the possibility of sharing the results to common sub-problems. The scheme shown in Figures 2.3–2.6 failed to work when elements were inserted or deleted from the sequence because the decomposition scheme did not break the problems into *common* sub-problems. A decomposition scheme that decomposes two similar problems in a way such that they share common sub-problems is called a *stable decomposition*. Whether a decomposition scheme is stable depends on what we consider *similar* problems; our definition of similar problems should operate at the abstract level and not depend on how the problems are represented. Creating data structures and algorithms with a stable decomposition is an interesting problem that is examined in detail and more formally in Section 4.

2.2 The Implementation of Function Caching

If we hope to use function caching to provide incremental computation, we must have an efficient implementation of function caching. Some previous implementations of function caching impose large overheads that make function caching only appropriate for combinatorial functions like the Fibonacci function. We report elsewhere solutions to a number of problems related to the efficient implementation of function caching [Pug88a] [Pug88b]. In our implementation, we found that function caching imposed about a 50% overhead on the speed of execution in situations where no hits were obtained. In empirical tests of applications such as incremental theorem proving, function caching provide overall real-time speed-ups of 4 to 6. An efficient function caching system must provide a method for calculating a hash keys for values and efficient, constant-time equality tests; we assume these are available in the rest of this paper. Fast equality tests can be provided by hashed consing [All78] or lazy structure sharing [Pug88b].

3 Randomized Data Structures and Probabilistic Time Bounds

The data structures we present in the Sections 5 and 6 are randomized (i.e., organized probabilistically). Algorithms that work with randomized data structures have good expected-time bounds and poor worst-case time bounds. In this section, we briefly present a unified method for analyzing both the expected performance and the probability distribution of the running times of algorithms.

A *random variable* has a fixed but unpredictable value and a predictable probability distribution and average. If X is a random variable, PROB$\{ X = x \}$ denotes the probability that X equals x and Prob$\{ X \le x \}$ denotes the probability that X is at most x. For example, if X is defined as the number obtained by throwing a perfect die, Prob$\{ X \le 3 \} = 1/2$. For conciseness, we sometimes refer to a property of the probability distribution of a random variable as a property of the random variable itself.

It is often preferable to find simple upper bounds on values whose exact value is difficult to calculate. In order to discuss upper bounds on random variables, we need to define a partial ordering on the probability distributions of random variables:

Definition 3.1 ($=_{prob}$ *and* $\le_{prob}$). Let X and Y be random variables. We define equality and a partial ordering on the probability distribution of random variables as follows:

$X =_{prob} Y$ iff $\forall\, x$, Prob$\{ X \le x \}$ = Prob$\{ Y \le x \}$ and
$X \le_{prob} Y$ iff $\forall\, x$, Prob$\{ X \le x \} \ge$ Prob$\{ Y \le x \}$. □

The algorithms we analyze in this paper have running times that are bounded by a negative binomial distribution.

Definition 3.2 (*negative binomial distributions* — $NB(s, p)$). Let s be a non-negative integer and p be a probability. The term $NB(s, p)$ denotes a random variable with the *negative binomial distribution* equal to the distribution of the number of failures seen before the s^{th} success in a series of random independent trials where the probability of a success in a trial is p. We define $NB(0, p) \equiv 0$ and $NB(s) \equiv NB(s, 0.5)$. □

Several well-known properties of the negative binomial distribution are [TK84]:

$$\text{avg}(NB(s,p)) = s(1-p)/p,$$
$$\text{variance}(NB(s,p)) = s(1-p)/p^2, \text{ and}$$
$$\text{Prob}\{ NB(s,p) = k \} = \binom{k+s-1}{s-1} p^s (1-p)^k.$$

If X and Y are two independent random variables, each bounded by a negative binomial distribution, we can calculate a probabilistic upper bound on $X + Y$:

$$X \le_{prob} c_1 + NB(s_1, p) \wedge Y \le_{prob} c_2 + NB(s_2, p)$$
$$\wedge\ X \text{ is independent of } Y$$
$$\Rightarrow X + Y \le_{prob} c_1 + c_2 + NB(s_1 + s_2, p).$$

If X and Y are *not* independent, we can still calculate an upper bound on the average of $X + Y$:

$$X \le_{prob} c_1 + NB(s_1, p) \wedge Y \le_{prob} c_2 + NB(s_2, p)$$
$$\Rightarrow \text{avg}(X + Y) \le \text{avg}(c_1 + c_2 + NB(s_1 + s_2, p)).$$

The probabilistic *big-Oh*

Big-Oh notation is usually defined by saying that

$$f(n) \text{ is } O(g(n)) \text{ iff } \exists\, M, n_0 \text{ s.t.} \forall\, n \text{ s.t. } n \ge n_0,\ f(n) \le M\, g(n).$$

This is adequate for analyzing algorithms that do not involve randomness; we introduce a new notation appropriate for analyzing algorithms involving randomness:

previously computed result $f(x)$ when computing $f(y)$. This will happen iff $x = y$. If R is a one-to-one function, this will be true whenever $R(x) = R(y)$. In this example, we could use a representation scheme in which a set is represented by a *sorted* list of the elements of the set.

It would be possible to side-step the requirement of unique representation by expecting our function caching implementation to match only arguments represented the same way. This would decrease the effectiveness of the cache, although in some circumstances the decrease might be small. If it were very difficult to provide data structures that had unique representations, we might wish to take this route. However, sections 5 and 6 provide data structures for sequences and sets that have the desired features.

4.3 Data structures and algorithms for incremental evaluation

In order to use function caching to solve quickly a new problem that is similar to a previous problem, the new problem must be broken down into sub-problems in a way such that solving the new problem involves solving sub-problems that were solved for the previous problem. Decomposing problems into sub-problems usually involves decomposing large data structures into smaller ones. We therefore wish to design data structures such that two *similar* values have *similar decompositions.*

Our definition of *similar* values depends on the type of similarities in which we are interested.

Definition 4.2 (*transformation*). A transformation on type τ is a function that maps a value of type τ (and possibly additional arguments) to a value of type τ. Transformations are not implemented; they are used only for discussing the similarity of abstract values. □

Example: One of the transformations on sequences we will be using in later examples is *changeFirst*. If *changeFirst* transforms x into x', then x and x' differ in only their first element. The meanings of other transformations we use is largely self-evident from their names. The *insert, delete* and *change* transformations on sequences reflect a change at any location. The *addElement* and *removeElement* transformations on sets reflect a difference of a single element.

Definition 4.3 (*distance between abstract values* — $tDistance_T(x, x')$). Let T be a set of transformations. We define $tDistance_T(x, x') = k$ iff k is the smallest integer such that there is a sequence of k transformations chosen from T that transforms x into x'. If no sequence of transformations from T transforms x into x', $tDistance_T(x, x')$ is undefined. Depending on T, this distance measurement may or may not be symmetric. □

Example: $tDistance_{\{changeFirst,\ insertBeforeFirst\}}([5, 1], [5, 3, 1]) = 2$ and $tDistance_{\{changeFirst,\ insertBeforeFirst\}}([5, 3, 1], [5, 1])$ is undefined. For sets A and A', $tDistance_{\{addElement,\ removeElement\}}(A, A') = |(A' - A) \cup (A - A')|$.

Definitions 4.4, 4.5 and 4.6 formalize the idea of *similar decompositions.*

Definition 4.4 (*decomposition scheme*). A *decomposition scheme* on type τ is a 3-tuple of functions $\langle$ *atomic, split, bound* $\rangle$ that have the types and properties shown below.

$atomic : \tau \rightarrow$ Bool,
$bound : \tau \rightarrow$ Nat,
$split : \tau \rightarrow \langle \tau, \tau \rangle$,
split is a one-to-one function,
$\neg\, atomic(x) \Rightarrow bound(x) > 0 \wedge split(x)$ is defined,
$\langle y, z \rangle = split(x) \Rightarrow$
$\quad bound(y) < bound(x) \wedge bound(z) < bound(x)$. □

Informally, $atomic(x)$ is true if it is impossible to further decompose x, $split(x)$ is the pair of values into which x is decomposed and $bound(x)$ is an upper bound on the number of times x can be decomposed.

To implement a decomposition scheme D on abstract values, we use a representation scheme R and decomposition scheme $D' = \langle$ *atomic', split', bound'* $\rangle$ such that D' is an efficient implementation of D for R (e.g., *split'* and *atomic'* are constant-time functions and $split'(x) = \langle y, z \rangle$ iff $split(R(x)) = \langle R(y), R(z) \rangle$).

Example: We can define a decomposition scheme *linkedLists* on sequences as

$atomic(S) \equiv |S| \leq 1$,
$bound(S) \equiv |S|$, and
$split([x_0, x_1, ..., x_{n-1}]) \equiv \langle [x_0], [x_1, ..., x_{n-1}] \rangle$.

Definition 4.5 (*decomposition* — $d_D(x)$). The *decomposition* of a value with respect to a decomposition scheme $D = \langle$ *atomic, split, bound* $\rangle$ is the set defined (recursively) by

$d_D(x) \equiv$ **if** $atomic(x)$ **then** $\{x\}$
$\quad$ **else** $\{x\} \cup d_D(y) \cup d_D(z)$
$\quad\quad$ **where** $\langle y, z \rangle = split(x)$ **fi**. □

Example: Using the decomposition scheme given in the example for Definition 4.4, $d_{linkedLists}([1, 2, 3]) = \{[1, 2, 3], [1], [2, 3], [2], [3]\}$.

Definition 4.6 (*distance between decompositions* — $dDistance_D(x, x')$). Let x and x' be values of type τ and D be a decomposition scheme on type τ.

$dDistance_D(x, x')$ denotes the distance from the decomposition of x to the decomposition of x' with respect to D: $dDistance_D(x, x') = |d_D(x') - d_D(x)|$. This distance measurement is not symmetric. □

Example: $dDistance_{linkedLists}([1, 4, 3], [1, 2, 3]) = |\{[1, 2, 3], [2, 3], [2]\}| = 3$.

Now that we have defined measurements that formalize the ideas of similar abstract values and of similar decompositions, we are ready to talk about *stable*

decompositions and the relevance of stable decompositions to incremental evaluation.

Definition 4.7 (*stable decompositions*). Let D be a decomposition scheme for values of type τ, T be a set of transformations on type τ, and $|x|$ be a measurement of the size of x.

The decomposition scheme D is said to be $O(f(|x'|))$ stable for T transformations if, for all x and x' in τ such that $tDistance_T(x, x')$ is defined, $dDistance_D(x, x')$ is $O(tDistance_T(x, x') f(|x'|))$.

If $dDistance_D(x, x')$ is $O_{prob}(tDistance_T(x, x') f(|x'|))$, D is said to be $O_{prob}(f(|x'|))$ stable for T transformations. □

Example: The *linkedLists* decomposition scheme for sequences (defined in the example for Definition 4.4) is $O(1)$ stable for {*deleteFirst, changeFirst, insertBeforeFirst*} transformations[1], $O(0)$ stable for {*deleteFirst*} transformations[2] and $O(n)$ stable for {*changeLast, deleteLast*} transformations[3] (i.e., totally unstable).

Theorem 4.1. Let $R : \sigma \rightarrow \tau$ be a representation function and T be a set of transformations on τ. Let $D = \langle$ *atomic, split, bound* $\rangle$ be a decomposition scheme for τ and $D' = \langle$ *atomic', split', bound'* $\rangle$ be a decomposition scheme on σ that is an implementation of D for R (e.g., $split'(x) = \langle y, z \rangle$ iff $split(R(x)) = \langle R(y), R(z) \rangle$) such that *atomic'* and *split'* are constant-time functions. Let g be a function on σ defined as

$$g(x) \equiv \textbf{if } atomic'(x) \textbf{ then } h_1(x) \textbf{ else } h_2(g(y), g(z)) \textbf{ where } \langle y, z \rangle = split'(x) \textbf{ fi}.$$

If
- D is $O(f(|x|))$ stable for T transformations,
- h_1 and h_2 can be computed in constant time,
- the results of all the recursive invocations of g involved in computing $g(x)$ are stored in the function cache, and
- function caching imposes only a constant-factor overhead,

then the time required to compute $g(x')$ is

$$O(tDistance_T(R(x), R(x'))\ f(|x|)).$$

If D is $O_{prob}(f(|x|))$ stable, the time required to compute $g(x')$ is $O_{prob}(tDistance_T(R(x), R(x'))\ f(|x|))$.

Proof. Computing $g(x')$ without function caching requires computing $g(y)$ for each y in $d_{D'}(x')$. Excluding the cost of performing recursive calls to g, each of these calls requires constant time (e.g., the total time to compute $g(x')$ without function caching is $O(|d_{D'}(x')|)$. The only calls to g that will not be found in the function cache are the calls with arguments from $d_{D'}(x') - d_{D'}(x)$. Because D' is defined as an implementation of D, $|d_{D'}(x') - d_{D'}(x)| = |d_D(R(x')) - d_D(R(x))|$. Since D is $O(f(|x|))$ stable for T transformations, $|d_D(R(x')) - d_D(R(x))|$ is $O(tDistance_T(R(x), R(x'))\ f(|x|))$. The argument follows similarly for the O_{prob} case. □

Of course, the idea of stable decompositions might be somewhat pointless if data structures fulfilling these criteria did not exist. We have developed efficient representations for sequences and sets that allow efficient applicative updates and have unique representations and stable decompositions. In Section 5, we present a representation for sequences that provides unique representations and is $O_{prob}(\log n)$ stable for {*insert, change, delete*} transformations. In Section 6, we present a representation for sets that provides unique representations and is $O_{prob}(\log n)$ stable for {*addElement, removeElement*} transformations.

4.4 Hash keys

The data structures we describe in Sections 5 and 6 are organized on the basis of the hash keys of the values stored in the data structures. The hash keys we need are the kind usable for extendible hashing [Meh84] (i.e., let $hash(x)$ be a hash function mapping U to $0..2^k-1$; if for all j such that $1 \leq j \leq k$, $hash(x)$ mod 2^j is a "good" hash function, then $hash(x)$ is acceptable for our purposes). The function caching implementation also requires hash keys, so these hash keys are already available.

5 A Decomposition and Representation Scheme for Sequences

This section describes a stable decomposition scheme for sequences (i.e., lists) and a scheme for representing sequences using tagged-tuples or S-expressions. We also describe algorithms that work with this representation: the function that splits a sequence requires constant-time; functions that access, change, delete and insert an element in a sequence of length n requires $O_{prob}(\log n)$ time; and the function that appends a sequence of length n to a sequence of length m requires $O_{prob}(\log n + \log m)$ time.

5.1 The chunky decomposition scheme

The chunky decomposition scheme is $O_{prob}(\log n)$ stable for {*insert, delete, change*} transformations and is based on the *levels* of elements of sequences, which are in turn based on the hash keys of elements.

---

[1] The sequence of transformations that produces the largest difference between the decompositions of x and x' is a series of insertions. If x' is the result of inserting k new elements at the front of x, then $tDistance_{\{deleteFirst, changeFirst, insertBeforeFirst\}}(x, x') = k$ and $dDistance_{linkedLists}(x, x') = 2k$.

[2] If $tDistance_{\{deleteFirst\}}(x, x')$ is defined, x' is a suffix of x and $dDistance_{linkedLists}(x, x') = 0$.

[3] If x' is identical to x except for the last element, $tDistance_{\{changeLast, deleteLast\}}(x, x') = 1$ and $dDistance_{linkedLists}(x, x') = |x'| + 1$.

Definition 5.1 (*level*(x)). The level of an element x, *level*(x), is the largest integer i such that *hash*(x) is a multiple of 2^i. □

The properties of hash functions stated in Section 4.4 guarantee that the levels of elements have the same probability distribution as $NB(1)$ (e.g., on average, half of the elements of a sequence are level 0, one quarter are level 1, and so on). We assume that no duplicate elements exist; the problems caused by duplicate elements are discussed in Section 5.2. Occurrences of only a relatively small number of duplicate elements has no significant effect.

Definition 5.2 (*chunky decomposition scheme*). The chunky decomposition scheme on sequences is defined as:

$$\begin{aligned}
&\mathit{atomic}(S) \equiv |S| \le 1,\\
&\mathit{bound}(S) \equiv |S|, \text{ and}\\
&\mathit{split}([\, x_0, \ldots, x_{n-1} \,]) \equiv \langle\, [\, x_0, \ldots, x_{i-1} \,], [\, x_i, \ldots, x_{n-1} \,] \,\rangle,\\
&\quad \textbf{where } i \textbf{ uniquely satisfies}\\
&\qquad 0 < i < n\\
&\qquad \wedge\, (\forall\, j \text{ s.t. } 0 < j < i,\ \mathit{level}(x_i) \ge \mathit{level}(x_j))\\
&\qquad \wedge\, (\forall\, j \text{ s.t. } i < j < n,\ \mathit{level}(x_i) > \mathit{level}(x_j)).
\end{aligned}$$ □

Informally, the chunky decomposition scheme divides a sequence $S = [\, x_0, \ldots, x_{n-1} \,]$ at the most *preferable break* in S. A *break* is a position between two elements in a sequence. The break immediately before element x_i is *preferable* to the break immediately before element x_j iff $\mathit{level}(x_i) > \mathit{level}(x_j)$ or ($\mathit{level}(x_i) = \mathit{level}(x_j)$ and $i > j$). The most preferable break therefore is immediately to the left of the rightmost element of S with maximal level (ignoring the first element of S). Figures 5.1 and 5.2 show the decompositions of two similar sequences according to this decomposition scheme.

Theorem 5.1 characterizes the subsequences that appear in the chunky decomposition of a sequence.

Theorem 5.1. Let $S = [\, x_0, x_1, \ldots, x_{n-1} \,]$. For all i, j such that $0 \le i \le j < n$, $[\, x_i, \ldots, x_j \,]$ appears in the decomposition of S iff $i = j$ or $\mathit{level}(x_i) > \max(\mathit{level}(x_{i+1}), \mathit{level}(x_{i+2}), \ldots, \mathit{level}(x_j)) \le \mathit{level}(x_{j+1})$ (to handle boundary conditions, assume that $\mathit{level}(x_0) = \infty$, and an element x_n exists and $\mathit{level}(x_n) = \infty$).

Proof. The sequence $[\, x_i, \ldots, x_j \,]$ appears in the decomposition of S iff the breaks immediately before x_i and before x_{j+1} are preferable to the breaks before $x_{i+1}, \ldots, x_j$. The break immediately before x_i is preferable to the breaks before $x_{i+1}, \ldots, x_j$ iff $\mathit{level}(x_i) > \max(\mathit{level}(x_{i+1}), \mathit{level}(x_{i+2}), \ldots, \mathit{level}(x_j))$. The break immediately before x_{j+1} is preferable to the breaks before $x_{i+1}, \ldots, x_j$ iff $\max(\mathit{level}(x_{i+1}), \mathit{level}(x_{i+2}), \ldots, \mathit{level}(x_j)) \le \mathit{level}(x_{j+1})$. □

Theorem 5.1 nicely describes why this decomposition scheme is stable: whether or not $[\, x_i, \ldots, x_j \,]$ appears in the decomposition of S depends solely on the levels of the elements $x_i, \ldots, x_j$ and x_{j+1}.

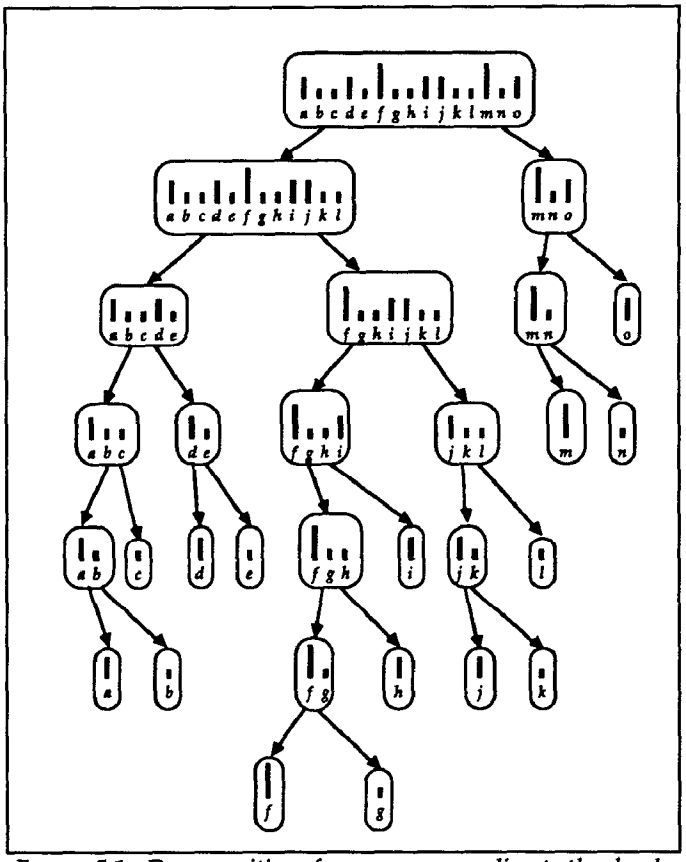

FIGURE 5.1 - Decomposition of a sequence according to the chunky decomposition scheme. The level of each element is indicated by the height of the bar above that element.

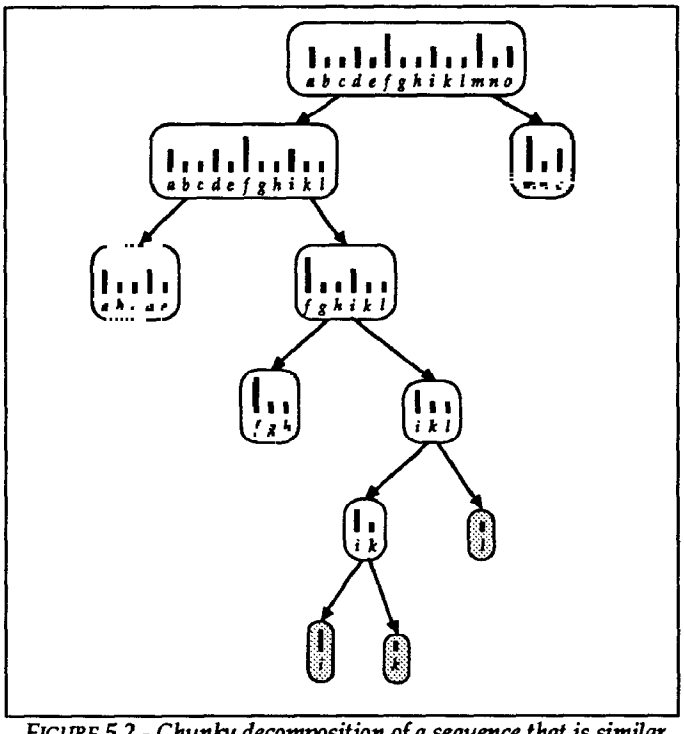

FIGURE 5.2 - Chunky decomposition of a sequence that is similar to the sequence in Figure 5.1. Subsequences that also appear in Figure 5.1 are shown in grey here and their decomposition is not shown further.

We now analyze the stability of this decomposition scheme with respect to {*insert, delete, change*} transformations.

Theorem 5.2. Let $S = [\ x_0, ..., x_{i-1}, x_{i+1}, ..., x_{n-1}\ ]$ and $S' = [\ x_0, ..., x_{n-1}\ ]$ (i.e., S' is the same as S except that a new element x_i has been inserted). The only sequences that appear in the decomposition of S' but not in the decomposition of S are those that include either x_{i-1} or x_i.

Proof. Let $[\ x_j, ..., x_k\ ]$ be a sequence not including x_{i-1} or x_i that appears in the decomposition of S' (i.e., $k < i-1$ or $i < j$). The presence of this sequence in the decomposition of S' does not depend on the level of x_i, and therefore it must also appear in the decomposition of S.

Theorem 5.3. Let $S = [\ x_0, ..., x_{n-1}\ ]$ and $S' = [\ x_0, ..., x_{i-1}, x_{i+1}, ..., x_{n-1}\ ]$ (i.e., S' is the same as S except that x_i has been deleted). The only sequences that appear in the decomposition of S' but not in the decomposition of S are those that include x_{i-1}.

Proof. Let $[\ x_j, ..., x_k\ ]$ be a sequence not including x_{i-1} that appears in the decomposition of S' (i.e., $k < i-1$ or $i < j$). The presence of this sequence in the decomposition of S' does not depend on the absence of x_i, and therefore it must also appear in the decomposition of S.

Theorem 5.4. Let $S = [\ x_0, ..., x_{n-1}\ ]$ and $S' = [\ x_0, ..., x_{i-1}, x_i', x_{i+1}, ..., x_{n-1}\ ]$ (i.e., S' is the same as S except that x_i has been replaced by x_i'). The only sequences that appear in the decomposition of S' but not in the decomposition of S are those that include either x_{i-1} or x_i'.

Proof. Let $[\ x_j, ..., x_k\ ]$ be a sequence not including x_{i-1} or x_i' that appears in the decomposition of S' (i.e., $k < i-1$ or $i < j$). The presence of this sequence in the decomposition of S' does not depend on the level of x_i', and therefore it must also appear in the decomposition of S.

Theorems 5.5 and 5.6 show that the chunky decomposition scheme is $O_{prob}(\log n)$ stable for {*insert, delete, change*} transformations.

Theorem 5.5. Let x_d be a designated element in a sequence $S = [\ x_0, ..., x_{n-1}\ ]$. The number of sequences containing x_d that appear in the decomposition of S is $O_{prob}(\log n)$.

Proof. Let $S_0, S_1, ..., S_{k-1}$ be the sequences that appear in the decomposition of S that include x_d, where $S_0 = [\ x_d\ ]$, $S_{k-1} = S$ and for all i, $0 \le i < k-1$, S_i is a either a proper prefix or a proper suffix of S_{i+1}.

Let $Left_i$ and $Right_i$ be defined such that $S_i = [\ x_{Left_i}, x_{Left_i+1}, ..., x_{Right_i-1}\ ]$. For all i, $0 \le i < k-1$, either $Left_{i+1} < Left_i$ and $Right_i = Right_{i+1}$ or $Left_{i+1} = Left_i$ and $Right_i < Right_{i+1}$.

If $Left_{i+1} < Left_i$, then $level(x_{Left_{i+1}}) > level(x_{Left_i})$. The number of distinct values in $[Left_0, Left_1, ..., Left_k] \le 1 + \max(\ level(x_0), level(x_1), ..., level(x_d)\ )$. From Theorem A.2, we derive the result that $\max(\ level(x_0), level(x_1), ..., level(x_d)\ ) \le_{prob} \lg d + NB(1)$.

The number of distinct values in $[Right_0, Right_1, ..., Right_k]$ is the number of non-decreasing elements in $[\ level(x_d), level(x_{d+1}), ..., level(x_{n-1})\ ]$. From Theorem A.3, we derive the result that this value is $\le_{prob} 1 + NB(1 + \lg(n-d)) + NB(1, 1/3)$.

Therefore, $k =$ (the number of of distinct values in $[Left_0, Left_1, ..., Left_{k-1}]$) + (the number of distinct values in $[Right_0, Right_1, ..., Right_{k-1}]$) − 1. Combining our results, we get

$$k \le_{prob} 1 + \lg d + NB(2 + \lg(n-d)) + NB(1, 1/3). \square$$

Theorem 5.6. The chunky decomposition scheme is $O_{prob}(\log n)$ stable for {*insert, delete, change*} transformations.

Proof. Let S and S' be sequences such that $tDistance_{\{insert, delete, change\}}(S, S') = k$. By Theorems 5.2, 5.3 and 5.4, there is a set D of no more than $2k$ elements from S' such that a sequence appears in the decomposition of S' but not of S only if it contains an element from D. Based on Theorem 5.5, the number of sequences appearing in the decomposition of S' that contain an element from D is $O_{prob}(|D| \log |S'|)$. $\square$

5.2 Duplicate elements

The reason we have assumed that sequences do not contain any duplicate elements is that we can then treat the levels of elements as independent random variables. For example, if a sequence simply contained n instances of an element x, all the elements would have the same level, and the decomposition rule would simply remove the rightmost element of a sequence. If no more than a small proportion of the elements in a sequence are duplicates, it should have no significant impact. We are pursuing research on a representation for sequences with many duplicate elements, but it is currently an open problem.

5.3 The chunky list representation scheme

In this section, we describe a representation scheme for sequences that provides unique representations and allows a sequence to be split according to the chunky decomposition rule in constant time. We also describe algorithms for appending the representation of two sequences and accessing, deleting, changing or inserting elements in the representation of a sequence; these algorithms all require $O_{prob}(\log n)$ time.

We define the chunky representation scheme by defining a inverse representation function R^{-1} that maps a sequence S to the S-expression that represents S. These S-expressions are presented as tagged tuples, as described in Section 4. Any S-expression in the range of R^{-1} is termed a *chunky list*.

Definition 5.3 (*chunky lists*). The inverse representation function R^{-1} that maps sequences to chunky lists is

$R^{-1}(S) \equiv$
  if S = [ ] **then** Empty( )
  else if S = [x] **then** Element(c)
    where c uniquely represents x
  else cList($R^{-1}(S')$, $R^{-1}(S'')$)
    where ⟨ S', S'' ⟩ = *split*(S).

The chunky list already reflects the chunky decomposition scheme, so no work is needed to decompose the representation of a sequence; the *split* function on chunky lists is constant-time. The implementation of *append* described in Figure 5.3 requires $O_{prob}(\log n + \log m)$ time to append the representations of two sequences of length n and m. The presentation of this algorithm uses the pattern-matching case statement that is typical of functional programming languages: free variables of a successively matched pattern are bound to the appropriate components of the value on which the case statement is discriminating. Asterisks (*) represent wild-cards.

We extend *level* so that, when applied to a chunky list representing a sequence S, it is the level of the first element of S. By caching the level of the first element in a sequence S with the data structure representing S, the *level* function remains a constant-time function (i.e., we change the chunky list representation scheme so that $R^{-1}([\ x_0, x_1, ..., x_{n-1}\ ])$ = cList($R^{-1}(S')$, $R^{-1}(S'')$, *level*(x_0)) where ⟨ S', S'' ⟩ = *split*([x_0, x_1, ..., x_{n-1}])). This change decreases the clarity of the algorithms slightly and is a fairly easy, mechanical change, so it has been omitted from the descriptions given below.

The algorithm is based on the idea of maintaining the invariants required by Definition 5.3. If S and T are both sequences of more than one element, the most preferable break in *append*(S,T) is either:

- the most preferable break in S,
- the break between S and T, or
- the most preferable break in T.

By using the rules given for the chunky decomposition rule, we can decide which of these breaks is most preferable.

Theorem 5.6. The time required by the algorithm of Figure 5.3 to append the representations of two sequences S and T is $O_{prob}(\log |S| + \log |T|)$.

Proof. Let S = [x_0, x_1, ..., x_{n-1}] and T = [y_0, y_1, ..., y_{m-1}]. Each recursive step in *append* involves stepping down a right branch of the data structure representing S or stepping down a left branch on the data structure representing T. The number of right steps in the data structure representing S is the number of sequences containing x_{n-1} that appear in the decomposition of S, which is $O_{prob}(\log |S|)$. The number of left steps in the data structure representing T is the number of sequences containing y_0 that appear in the decomposition of T, which is $O_{prob}(\log |T|)$. □

```
append(S, T) ≡
  case ⟨ S, T ⟩ of

    ⟨ Empty( ), * ⟩ ⇒ T
    ⟨ *, Empty( ) ⟩ ⇒ S
    ⟨ Element( * ), Element( * ) ⟩ ⇒ cList( S, T )

    ⟨ Element( * ), cList( T0, T1 ) ⟩ ⇒
      if level(T0) ≤ level(T1)
          { then break before T1 most preferable }
          then cList( append(S, T0), T1 )
          { else break before T most preferable }
          else cList( S, T )

    ⟨ cList( S0, S1 ), Element( * ) ⟩ ⇒
      if level(S1) ≤ level(T)
          { then break before T most preferable }
          then cList( S, T )
          { else break before S1 most preferable }
          else cList( S0, append(S1, T) )

    ⟨ cList( S0, S1 ), cList( T0, T1 ) ⟩ ⇒
      if level(S1) ≤ level(T1) and level(T0) ≤ level(T1)
          { then break before T1 most preferable }
          then cList( append(S, T0), T1 )
      else if level(S1) ≤ level(T0) and level(T1) < level(T0)
          { then break before T most preferable }
          then cList( S, T )
      { else break before S1 most preferable }
      else cList( S0, append(S1, T) )
```

FIGURE 5.3 – Description of algorithm to append two sequences represented as chunky lists.

We define functions *length*, *first* and *last* such that *length*(S) is the number of elements in the sequence represented by S, *first*(S, i) is the chunky list representing the first i elements in the sequence represented by S, and *last*(S, i) is the chunky list representing the last i elements in the sequence represented by S. The function *length* is cached with the data structures, as described for the level function, so that it is a constant-time function. We can implement *first* and *last* as shown in Figure 5.4. Also described in Figure 5.4 are functions to access, change, insert, or delete elements. Each of these operations requires $O_{prob}(\log n)$ time.

```
first(S, i) ≡
  If i = 0 then Empty( )
  else if i = length(S) then S
  else case S of
       cList( S0, S1 ) ⇒
            if i ≤ length(S0) then first(S0, i)
            else cList( S0, first(S1, i – length(S0) ) )

last(S, i) ≡
  If i = 0 then Empty( )
  else if i = length(S) then S
  else case S of
       cList( S0, S1 ) ⇒
            if i ≤ length(S1) then last(S1, i)
            else cList( last(S0, i – length(S1)), S1 )

access(S, i) ≡
  case last(first(S, i+1), 1) of
       Element( x ) ⇒ x

change(S, x, i) ≡
  append(first(S, i),
       append(Element( x ), last(S, length(S) – i – 1)))

delete(S, i) ≡ append(first(S, i),
                      last(S, length(S) – i – 1))

insert(S, x, i) ≡
  append(first(S, i),
       append(Element( x ), last(S, length(S) – i)))
```

FIGURE 5.4 – Description of algorithms to extract the first and last i elements of a sequence and to access, change, delete and insert before element i of a sequence.

6 A Decomposition and Representation Scheme for Sets

This section describes a stable decomposition scheme for sets and a scheme for representing sets using tagged-tuples or S-expressions. We also describe algorithms that work with this representation and give time bounds for these algorithms.

The algorithms and representations we describe use annotated sets: a set annotated with a string of binary digits (possibly null). For example, $\langle S, L \rangle$ is the set S annotated with the string L. If $\langle S, L \rangle$ is an annotated-set value, then for all elements x in S, L is a suffix of the $hash(x)$. This label encodes an invariant that is maintained by our set algorithms; making it explicit simplifies our exposition. A set S is represented by the annotated set value $\langle S, \varepsilon \rangle$ (where ε is the null string). The predicate $suffix(L, k)$ is true iff L is a suffix of the binary representation of k. The term *hashBits* denotes the number of bits in a hash key (i.e., $hash(x) \in 0..2^{hashBits} - 1$). Since our set algorithms work with annotated sets, we define our decomposition scheme on annotated sets as shown below.

Definition 8.1 (*decomposition scheme for annotated-sets*). The decomposition scheme for annotated sets is defined as:

$$atomic(\langle S, L \rangle) \equiv |\{ hash(x) \mid x \in S \}| \leq 1,$$
$$bound(\langle S, L \rangle) \equiv hashBits - |L|, \text{ and}$$
$$split(\langle S, L \rangle) \equiv \langle \langle \{ x \in S \mid suffix(0L, hash(x)) \}, 0L \rangle, \langle \{ x \in S \mid suffix(1L, hash(x)) \}, 1L \rangle \rangle. \square$$

The representation scheme we use for annotated sets can be described by defining the inverse representation function R^{-1} as shown below. Let r^{-1} be the inverse representation function for a representation scheme in which sets are represented by a sorted list of the elements of the set.

$$R^{-1}(\langle S, L \rangle) \equiv$$
  if $S = \varnothing$ then EmptySet(L)
  else if $atomic(\langle S, L \rangle)$ then AtomicSet($r^{-1}(S), L$)
  else Pair($R^{-1}(\langle S', 0L \rangle), R^{-1}(\langle S'', 1L \rangle), L$),
    where $\langle \langle S', 0L \rangle, \langle S'', 1L \rangle \rangle = split(\langle S, L \rangle)$

This representation scheme is equivalent to using *binary hash tries*. Binary hash tries use a binary trie [Knu73] data structure based on the hash keys of the elements. Atomic annotated sets that appear in the decomposition of $\langle S, \varepsilon \rangle$ appear as leaf nodes in the trie representing $\langle S, \varepsilon \rangle$ and non-atomic annotated sets appear as interior nodes. Figure 8.1 shows the representation of a sample set, using the hash keys given in Table 8.1.

The appropriate divide and conquer algorithms for calculating the union, intersection or difference of annotated-sets (or testing to see if one annotated set is a subset of the other) are fairly obvious. For example, for set union:

$$\langle A, L \rangle \cup \langle B, L \rangle = \langle \langle A', 0L \rangle \cup \langle B', 0L \rangle, \langle A'', 1L \rangle \cup \langle B'', 1L \rangle \rangle$$
where $split(\langle A, L \rangle) = \langle \langle A', 0L \rangle, \langle A'', 1L \rangle \rangle$ and
  $split(\langle B, L \rangle) = \langle \langle B', 0L \rangle, \langle B'', 1L \rangle \rangle$.

The algorithm for performing set union using this representation of annotated sets is described in Figure 8.2. One point to note is that our algorithms make use of unique representations and constant-time equality tests. For example, when computing $\langle A, L \rangle \cup \langle B, L \rangle$, if $A = B$ we can immediately return $\langle A, L \rangle$ as the answer.

Pardo [Par78] suggested an equivalent data structure for representing sets. He did not consider the application of this representation for incremental computation, and his analysis was more complicated than ours and achieved poorer results. Part of the reason for his poorer results is that he did not make use of the ability to perform constant-time equality tests between sets provided by this representation.

If hash keys are chosen to be sufficiently long, atomic sets will very rarely contain more than a single element;

therefore, an atomic set S can be efficiently represented by a sorted list of the elements of S.

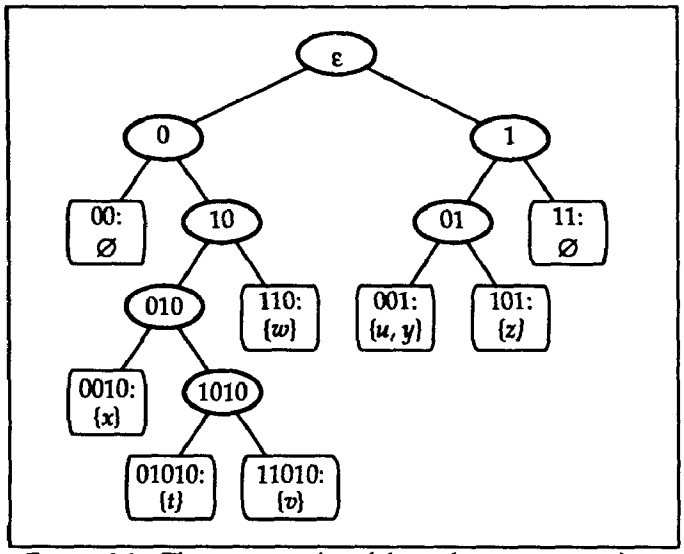

FIGURE 8.1 – The representation of the set {t, u, v, w, x, y, z} as a binary hash trie, using the hash keys shown in Table 8.1.

element	5-bit hash key
t	01010
u	01001
v	11010
w	00110
x	10010
y	01001
z	10101

TABLE 8.1 – Hash keys of the elements in the set in Figure 8.1

6.1 Set operations

The algorithms for performing set union are described in Figure 8.2. Other standard set operations such as intersection, difference and subset tests can be implemented in a method similar to the method used for set union, and the same time analysis holds.

6.2 Analysis

The complete analysis of this set representation is somewhat complicated and beyond the scope of the paper. In this section, we summarize the results presented in [Pug88b].

Non-incremental set operations

The time to compute A *op* B, where $op \in \{\cup, \cap, -, \subseteq\}$, is bounded by

$$O_{prob}(\,|S_2|\,(1+\log(|S_3|/|S_2|))\,)$$

where S_2 and S_3 are the middle and largest of the sets $A-B$, $B-A$, $A \cap B$. If the sets A and B differ in only k elements, the time required to calculate A *op* B is therefore only

$$O_{prob}(\,k\,(1+\log(|A \cap B|/k))\,).$$

```
union(X, Y) ≡ { set union }
  case ⟨ X, Y ⟩ of
    ⟨ AnnotatedSet( A ), AnnotatedSet( B ) ⟩
      ⇒ aUnion(A, B)

aUnion(X, Y) ≡ { annotated-set union }
  if X = Y then X
  else if readyForAtomicUnion(X,Y)
    then atomicUnion(X, Y)
  else case ⟨ splitAtomic(X), splitAtomic(Y) ⟩ of
    ⟨ *, EmptySet( L ) ⟩ ⇒ X
    ⟨ EmptySet( L ), * ⟩ ⇒ Y
    ⟨ Pair( A0, A1, L ), Pair( B0, B1, L ) ⟩ ⇒
      Pair( aUnion(A0, B0), aUnion(A1, B1), L)

splitAtomic(X) ≡
  case X of
    EmptySet( L ) ⇒ X
    Pair( *, *, * ) ⇒ X
    AtomicSet( B, L ) ⇒
      if suffix(0L, hash(B) )
        then Pair( AtomicSet( B, 0L ),
                   EmptySet( 1L ), L )
        else Pair( EmptySet( 0L ),
                   AtomicSet( B, 1L ), L )

readyForAtomicUnion(X, Y) ≡
  case ⟨ X, Y ⟩ of
    ⟨ AtomicSet( A, L ), AtomicSet( B, L ) ⟩
        ⇒ hash(A) = hash(B)
    ⟨ *, * ⟩ ⇒ false

atomicUnion(X, Y) ≡
  case ⟨ X, Y ⟩ of
    ⟨ AtomicSet( A, L ), AtomicSet( B, L ) ⟩ ⇒
      AtomicSet( PrimitiveUnion(A, B), L )
```

FIGURE 8.2 – A description of the algorithm for set union.

Decomposition stability

The decomposition scheme we have described for annotated sets is $O_{prob}(\log n)$ stable for (*addElement*, *removeElement*) transformations. This would give us fairly good time bounds for incremental set operations, but we can do even better, as shown in the next two items.

Increment membership tests

Assume the computations involved in determining if $x \in S$ are stored in the cache. Define ΔS to be $(S' - S) \cup (S - S')$. The time required to determine if $x \in S'$ is bounded by $O_{prob}(\log |\Delta S|)$ if $x \notin \Delta S$ and by $O_{prob}(\log |S'|)$ otherwise.

Incremental set operations

Assume that the computations involved in computing A *op* B are stored in the cache and consider computing A' *op* B', where A' is similar to A and B' is similar to B. Let S_1, S_2 and S_3 be the smallest, middle and largest of the sets $A' - B'$, $B' -$

A', $A' \cap B'$. Let $\Delta Both = (A'-A) \cup (A-A') \cup (B'-B) \cup (B-B')$. The time required to compute A' *op* B' is bounded by

$$O_{prob}(\ |\Delta Both - (S_1 \cup S_2)| \log(|S_2| / |\Delta Both|) + |\Delta Both \cap (S_1 \cup S_2)| \log(|S_3| / |\Delta Both|)).$$

6.3 Similar abstract data types

Representations and algorithms for finite functions (i.e., symbol tables), priority queues and bags can be based on the idea of binary hash tries used above for sets. A more detailed presentation of the application of binary hash tries to these data structures is described in [Pug88b].

7 A Comparison with Incremental Attribute Grammar Evaluation

Our motivation in examining this approach was to develop techniques for incremental evaluation in situations where attribute grammar and dependency graph schemes are unusable. How does function caching compare against incremental dependency graph evaluation on those applications for which incremental dependency graph evaluation does work well?

Graph evaluation and functional programs are sufficiently different that it is difficult to make any sort of direct comparison between the two approaches. For the case of attribute grammars, however, we can make a direct comparison. Katayma has described a method for translating an attribute grammar G into a set of recursive functions $F(G)$ [Kat84]. We can then compare the incremental performance of the Reps optimal attribute grammar propagation scheme [Rep82] on G with the incremental performance of using function caching on $F(G)$. The material in this section assumes the reader is somewhat familiar with attribute grammars and optimal attribute grammar evaluation.

The process of converting an attribute grammar into a set of recursive functions is fairly straight forward. Let N be a nonterminal in a grammar G. For each synthesized attribute s of N, we define a corresponding function s_N. Let r be a node of type N in a tree T. Let T_r be the subtree rooted at r. The function s_N is defined such that in a consistently attributed tree, $r.s = s_N(T_r, r.i_1, r.i_2, r.i_3..., r.i_m)$, where the set $\{i_1, i_2, i_3... i_m\}$ is the set of inherited attributes of r that s might possibly depend on (*i.e.*, $\{i_1, i_2, i_3... i_m\} = \{ i \in I(N) \mid (i, s)$ *is an edge of* $IO[N] \}$).

The process of translating an attribute grammar is fairly easy and can be mechanized [Kat84], although we will not go into the details of the translation process here. Because we plan to make use of function caching, we can avoid some of the complexities introduced by Katayama. Figure 7.1 illustrates the translation from attribute grammars to recursive functions using the standard example of the semantics of binary numerals. The recursive functions are shown with a syntax that is similar to that of PROLOG in order to make the correspondence between the attribution rules and the functions more obvious. Some additional mechanical manipulation can combine the multiple definitions given here for each s_N into a single function with a case statement. Efficient evaluation of the functional program produced relies on the use of function caching.

---

p_0: *Number* → *Integer*
Number.value = *Integer.value*
Integer.scale = 0
| value$_{Number}$(p_0(Integer)) = value$_{Integer}$(Integer, 0)

p_1: Integer → Integer Bit
Integer$_1$*.value* = *Integer*$_2$*.value* + *Bit.value*
Integer$_2$*.scale* = *Integer*$_1$*.scale* + 1
Bit.scale = *Integer*$_1$*.scale*
| value$_{Integer}$(p_1(Integer, Bit), scale) = value$_{Integer}$(Integer, scale+1) + value$_{Bit}$(Bit, scale)

p_2: Integer → Bit
Integer.value = *Bit.value*
Bit.scale = *Integer.scale*
| value$_{Integer}$(p_2(Bit), scale) = value$_{Bit}$(Bit, scale)

p_3: Bit → **0**
Bit.value = 0
| value$_{Bit}$(p_3(**0**), scale) = 0

p_4: Bit → **1**
Bit.value = $2^{Bit.scale}$
| value$_{Bit}$(p_4(**1**), scale) = 2^{scale}

FIGURE 7.1 - An sample attribute grammar and its translation to a set of recursive functions. The grammar defines the integer equivalent of a binary numeral.

Comparing function caching with incremental attribute grammar evaluation

We will now compare the performance of function caching and incremental attribute grammar evaluation.

Theorem 7.1. Let T be a tree consistently attributed according to an attribute grammar G. Define the set *Affected* to be the set of attributes that receive a new value as a result of a subtree replacement at a node *new* (as in Reps's discussion). Define *path_to_root* to be the set of nodes in T that are an ancestor of *new* (define *path_to_root* so that it includes *new*). Let *New_Applications* be the set of function applications that need to be computed and will not be found in an infinite cache when using function caching and $F(G)$. Then,

$$|New\_Applications| \text{ is } O(|Affected| + |path\_to\_root|).$$

Proof. Define the set *I_Affected_Nodes* to be the set of nodes r in T such that an inherited attribute of r receives a new value as a result of a change. Since each attribute is an inherited attribute of only one node, $|I\_Affected\_Nodes| \leq |Affected|$.

Define *Needed(T)* to be the set of function calls needed to evaluate the function calls from $F(G)$ corresponding to synthesized attributes of the root of T. For a function call f, let *root(f)* be the root of the subtree that is the first argument of f (e.g., if f is a function call whose first argument is T_r, $root(f) = r$). Since the number of synthesized attributes at a node is bounded by a constant based on the size of the grammar, for all nodes r in T, the size of $\{f \mid f \in Needed(T) \wedge root(f) = r\}$ is bounded by a constant.

In a function caching implementation, we cannot perform destructive editing operations. To perform an edit operation on a tree T, we create a new tree T' and evaluate the functions corresponding to the synthesized attributes of the root of T'. The nodes of T' that cannot be reused from T are those in *path_to_root*.

The only functions that need to be computed are those that were not computed previously. Therefore, $New\_Applications \subseteq \{f \mid f \in Needed(T') \wedge root(f) \in I\_Affected\_Nodes \cup path\_to\_root\}$. Therefore, $|New\_Applications|$ *is* $O(|I\_Affected\_Nodes| + |path\_to\_root|)$ which is $O(|Affected| + |path\_to\_root|)$. □

How well have we done? For very narrow and stringy trees, this would be very bad. In some situations, a change in a tree of size n would involve a length n path to root. Several points come to mind however. First, we feel that this is an argument against narrow stringy trees, not against function caching. Typically, program editors have used a left or right recursive format for sequences such as statement lists. The data structures we have presented in this paper, or many other data structures, allow lists of length n to be manipulated as trees of depth $\log n$. Second, the overhead of $|path\_to\_root|$ is partially associated with the fact that the conversion process is only appropriate for attribute grammars that only produce information at the synthesized attributes of the root of the syntax tree; for such attribute grammars, function caching has the same asymptotic time bounds as attribute grammars whenever a change to the syntax tree causes a change in the output of the algorithm.

We should also note that new research in incremental dependency graph evaluation has improved and extended Reps's algorithms in new directions [HT86] [Hoo87] [ACR+87]. Even if we wanted to claim that we could do as well as Reps's original algorithm, this would not say much about function caching verses attribute grammar or dependency graph techniques, since many new results have improved on Reps's original results.

8 Conclusions

Function caching and stable decompositions provide a new paradigm for incremental evaluation. The essential difference between this paradigm and techniques based on incremental dependency graph analysis is that our approach is based on reusing the previously computed solution to any similar problem; incremental dependency graph techniques are based on the idea of updating a specific previous problem. In some situations, we can predict that subproblems similar to previous subproblems will arise, but we can't establish *apriori* an appropriate one-to-one correspondence between new subproblems and previous subproblems. In situations such as this, function caching can provide incremental evaluation and incremental dependency graph techniques seem unusable.

There is no reason why these two paradigms can not be profitably combined. In many incremental attribute grammar systems, the attribute equations are specified using recursive functions. If function caching is performed on the calls made by these functions, we can use attribute grammar and dependency graph techniques when they are appropriate and function caching when they are not.

Unique representations, combined with a method for constant-time equality testing of concrete values, allows constant-time equality testing of abstract values. For many abstract data types, efficient representations that allow constant-time equality testing have been open questions. Sassa and Goto [SG76] describe a set representation that allows constant-time equality tests. However, in their representation, any set operation takes time at least proportional to the size of the set being created (e.g., to compute $S + \{x\}$ requires $O(|S|)$ time). A linked list offers unique representations for sequences, but modifying element i of a sequence requires $O(i)$ time.

The idea of a stable decomposition also has implications outside of incremental evaluation. Typically, if two values have similar decompositions they have equal subcomponents. When combined with a technique such as hashed consing, this allows two representations of similar values to share storage. Reps discusses using sharable 2-3 trees for sharing storage between similar values in his thesis [Rep84]. However, using his methods, two values can share storage only if they are similar in a "computation-oriented" sense, rather than in a "value-oriented" sense (e.g., if the set B has been calculated as $A + \{x\}$, the sets A and B can share storage; if the set B happens to be equal to $A + \{x\}$, but was not calculated that way, it might not be possible for the sets to share storage). Reps noted as an open problem a scheme that allows "value-oriented" similar values to share storage; we have closed this problem for sets and sequences without duplicate elements.

Appendix

This appendix includes the statement of two theorems, proved in [Pug88b], that are used in the analysis of chunky lists. These results concern collections of random variables, each bounded by a negative binomial distribution.

Theorem A.2 Let p be a probability, and $X_0, X_1, \ldots, X_{n-1}$ be independent random variables, each bounded by $NB(1, p)$.

$$\max(X_0, X_1, \ldots, X_{n-1}) \leq_{prob} \log_{1/(1-p)} n + NB(1, p)$$

Theorem A.3. Let p be a probability, and $X_0, X_1, ..., X_{n-1}$ be independent random variables, each bounded by $NB(1, p)$. Let ND be the set of non-decreasing elements in the sequence $[X_0, X_1, ..., X_{n-1}]$, which is equal to $\{ X_i \mid \max(X_0, X_1, ..., X_i) \leq X_i \}$.

$$| ND | \leq_{prob} 1 + NB(\log_{1/(1-p)} n, p) + NB(1, p^2/(1-p+p^2)) + NB(1, 1-p)$$

References

[ACR+87] B. Alpern, A. Carle, B.Rosen, P. Sweeney, and K. Zadeck. *Incremental evalaution of attributed graphs.* Technical Report RC 13205, IBM, Thomas J. Watson Research Center, Yorktown Heights, New York 10598, October 1987.

[All78] John Allen, *Anatomy of LISP*, McGraw Hill Book Company, NY, 1978.

[DRT81] Alan Demers, Thomas Reps and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. *PROC of the Eighth POPL*, pages 105-116, 1981.

[Hoo86] Roger Hoover, Dynamically bypassing copy rules in attribute grammar. *PROC of Thirteenth POPL*, pages 14-25, 1986.

[Hoo87] Roger Hoover, *Incremental Graph Evaluation.* Ph.D. Thesis, Cornell University, 1987.

[HT86] Susan Horowitz and Tim Teitelbaum, Generating editing environments based on relations and attributes. *TOPLAS*, pages 577-608, 1986.

[Kat84] Takuya Katayama. Translation of attribute grammars into procedures. *TOPLAS*, 6(3):345-369, July 1984.

[Knu73] Donald Knuth, Sorting and Searching, *The Art of Computer Programming*, Vol. 3, 1973.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984.

[Mic68] D. Michie. "Memo" functions and machine learning. *Nature*, (218):19-22, April 1968.

[Par78] Luis Isidoro Trabb Pardo, *Set Representation and Set Intersection*, Ph.D. thesis, Stanford University, 1978

[Pug88a] William Pugh. An Improved Cache Replacement Strategy for Function Caching. *PROC of the ACM Conf on Lisp and Functional Programming*, pages 269-276, 1988.

[Pug88b] William Pugh, *Incremental Computation and the Incremental Evaluation of Functional Programming.* Ph.D. Thesis, Cornell University, 1988.

[RA84] Thomas W. Reps and Bowen Alpern. Interactive proof checking. *PROC of the Eleventh POPL*, pages 36-45, 1984.

[Rep82] Thomas Reps, Optimal-time incremental semantic analysis for syntax-directed editors. *PROC of the Ninth POPL*, pages 169-176, 1982.

[Rep84] Thomas W. Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge Massachusetts, 1984.

[SG76] M. Sassa and E. Goto, A hashing method for fast set operations. *Inf. Proc. Let.* 5(2):31-34, June 1976.

[TK84] H. Taylor and S. Karlin. *An Introduction to Stochastic Modeling*, Academic Press, Orlando, Florida, 1984.

[YS88] D. Yellin and R. Strom. INC: a Language for Incremental Computations. *SIGPLAN '88 Conference on Programming Language Design and Implmentation*, pages 115-124, 1988.