Algebraic Data Types for Delta Encoding

Andrew Kalenda[*]

Department of Computer Science, San José State University

Abstract: Delta encoding is an established practice of representing data in terms of a delta — its difference to other data — for the purpose of more compactly expressing that data in sum. It is relevant to such applications as source code management, HTTP data transfer, and incremental storage updates. To minimize size, deltas are streamed as a single predetermined type (i.e. string deltas), and deltas of another type (i.e. syntactic tree deltas) are streamed separately. This research project explores construction of deltas as algebraic data types able to accommodate any number of delta types through a single data stream; algebraic refactoring of deltas; its potential efficacy and ease of use; and its use in at least one application.

1 Introduction

In my last year of undergraduate studies, I had a group project in which we programmed a graphics engine. Typically in group projects, the work is divided into modules which programmers work on separately, making edits to different files, or at the very least different regions of the same file. This group project was different, though; our paired programming had become fluid to the extent that, although on separate laptops, we were able to make coordinate changes not just to the same function in the same file, but the same loop. We rapidly sent changes back and forth between our git repositories, without error. It was some of the most fun I have had coding, but much of it was spent pushing and pulling commits.

Since then, I've often found myself contemplating how technology could facilitate that kind of paired programming. I started looking more closely at Google Docs, an in-browser cloud-based document writer in which multiple users can simultaneously make edits and see what others are typing, what they have selected, and so forth in real-time. Could we have the same kind of real-time editing for an IDE? Absolutely!

My thinking extended further along this line. Many people can be working on the same code base, making a high volume of edits. On the other hand, an IDE will know much more about the nature of edits being made than a simple text editor. Where a plain text editor might only see various lines being altered, an IDE can know that a particular symbol – like a variable name – is being renamed. There is the potential to significantly reduce necessary data transmission by taking advantage of this kind of information.

To begin with, we have the classic practice of encoding changes in terms of deltas, widely used in the HTTP protocol[2, 4]. Simply put, deltas describe the differences between data; knowing what data at one point in the sequence is, and knowing the difference between it and the next, we can reconstruct that next data point. Primarily, this is useful for compressing data transmission at a small computational cost, which is exactly what is desired for the online IDE idea. Section 2 explains delta encoding further.

But how does the aforementioned IDE knowledge translate into savings in these deltas? As in the case given, where it knows about a symbol rename, it has the option of encoding a delta

[*] andrew.kalenda@gmail.com

in terms of either a delta on the plain text file, or a delta on the program's abstract syntax tree. The second option is very likely to be much more compact. Thus, we can use different *types* of deltas.

From this, an idea formed from the vapor of my mind: If the delta types are variants, why not express them as an algebraic data type? Then the delta is always of one type which we can construct to accommodate any combination of possible deltas we could desire. Perhaps these deltas could even be reconstructed dynamically to compress data further – all while maintaining strong program safety. Section 3 explains algebraic data types in greater detail.

Without variants and/or algebraic data types, handling multiple types of deltas is still possible. A web app could simply have multiple streams to the server, each specified for a particular type of delta. The project's intent is to combine these streams into a single stream of algebraically typed deltas.

My hypothesis is that, although a stream of algebraically typed deltas may require extra bytes in order to tag variants, the ability to safely combine sets of changes into cleaner and more efficient deltas would lead to less data transmission and fewer packets. I further hypothesize that it may require no extra bytes whatsoever to encode, as deltas are already essentially command expressions, and we need only add more commands. I also hypothesize as these commands bear resemblance to abstract syntax – for which ADTs are particularly well-suited[*citation needed*]– an algebraic approach would lead to a cleaner, safer, more understandable implementation.

With this hypothesis, an experiment is necessary. Currently the experiment is pending, and will be detailed along with its results in a future version of this paper. The original motivation, an online IDE, is beyond the scope of this experiment. Instead, the experiment will depend on a minimal application to provide proof of concept.

The plan is for this to be a web app, a simple text editor that handles two delta types: The first being standard changes to the text file based on line and column; the second being find-and-replace changes. These deltas will then be sent to two servers. The first will be sent deltas as plain data types through separate streams. The second will be sent deltas as an algebraic data type through a single stream. The two data streams will be metrically compared for volume of data and number of packets. The two implementations will also be compared by their "soft" attributes, such as clarity and apparent safety.

From this point, we can potentially explore optimizations available for each method, and perhaps continue the experiment by implementing these optimizations if they seem worthwhile.

2 Delta Encoding

2.1 Definitions

Delta encoding is a protocol for **data differencing**. The contents of a file are expressed not in absolute terms, but relative to the contents of another file. By knowing the contents of file A and the difference in content between files A and B, we are able to construct the contents of file B. A **delta** is a set of commands that describe one of these differences: For example, a delta might read, “at index 7 append the string ’foo’.”

Directed deltas require, or specify, these differences in a specific order. A delta is described as the *set-theoretical difference* in content between a file F and its predecessor: $\Delta F_i = F_i - F_{i-1}$. (This is also called the *relative complement* of F_{i-1} in F_i.) Consequently, we can describe any file F in the sequence as a construction of an initial file F_0 and a series of deltas "leading" to it: $F_i = F_0 + \Delta F_1 + \Delta F_2 + ... + \Delta F_i$.

Undirected or **symmetric** deltas describe paired files in terms of their set-theoretic symmetrical difference. The symmetric difference is simply the union of the relative complements: $\Delta F_{(a,b)} = (F_a - F_b) \bigcup (F_b - F_a)$. Thus the ordering does not matter, as the delta can be used to construct either file from the other.

2.2 Relevance

Delta encoding is often called **delta compression**, as it is almost always used to reduce the amount of data that needs to be stored in a series of files where the differences between those files is typically much smaller than the files in their entirety.

One example of this would be source code management systems, in which all the versions of a file are maintained. The deltas between one version and another are stored, allowing any version in the series to be reconstructed. Because the delta between two versions may often be as slight as changing a single line out of hundreds of lines in the source, this saves a great deal of space.[*citation needed*]

A second example is in video encoding. Instead of encoding deltas between files, we can encode deltas between frames in the video. Each frame is a sequential image, and the difference between one frame and the next may be a slight shift in the colors of a subset of pixels. Storing the frame deltas can significantly reduce the data required for a video.[*citation needed*]In the case of a cartoon, where images are often largely static with only a small changing sub-frame (i.e. the mouth of a speaking character), the space savings could be immense.

Another application is in cryptography and information security. In this case, delta encoding is used not to compress data, but to obfuscate data and confuse malign intellects. That data does not necessarily need to be a file, but any important value used for encryption. Examples include a private key, indices in a table, cypher text, or even the deltas for a series of deltas.[1] Recall from our definitions that $F_i = F_0 + \Delta F_1 + \Delta F_2 + ... + \Delta F_i$. A malicious eavesdropper may be able to intercept one or more deltas, but unless they are able to intercept *all* deltas, they will be unable to reconstruct the actual data. Furthermore, if the data thus changed is used to encrypt other data, then those encryption algorithms’ behavior will constantly be shifting, and thus be that much more difficult to hack.

The final example we will give – although it’s far from the last we *could* give! – is in the field of evolutionary algorithms. These are algorithms that mimic natural systems, such as genetic mutation or economic markets.[*citation needed*]A change to a single value can propagate through the system, creating changes in the system state on a potentially exponential order. In studying and storing the evolution of these systems, it can be beneficial to store deltas for those changes in the system which cannot be deterministically reconstructed.

2.3 Code examples

This section is concerned with implementation, and its development will be seen in a future draft contingent on our work in Project 2. It will give a very simple algorithm for encoding a delta between two strings, and then will briefly discuss the classic Myer's diff[3] and our intent to use the google-diff-match-patch library. We are as yet unclear which algorithm would be best for our project's demonstration purposes, versus which would be best for an actual application.

```
        /* Simple string delta encoder pending
         */
```

3 Algebraic Data Types

3.1 Definitions

A **data type**, or simply **type**, defines the set of possible values for a datum. It also implies how a datum can be stored, used, and construed.

A **variant type** is one in which the type itself is variable. E.g. It can be one of several types. If **tagged**, such as the variants in ML, Haskell, Scala, or F#, then it will assume a specific type when a value is stored. If **untagged**, such as C/C++'s `union`, then it will assume a specific type when a value is retrieved.

A **composite type** is one which is built out of combinations of data. It is an essential programming tool, expressed as structs, classes, objects, records, and so on.

A **recursive type** is any self-referential data type. In the case of a recursive variant type, it must also be **inductive**, where the variant's *actual* type will eventually resolve to a non-recursive type as its base case. This is not the case for a recursive composite type, as its type is invariant and does not need to be resolved.

An **algebra** is a set of *operations* that describe how data of a type relate to one another. Any algebraic expression can be considered as an **n-ary operation**, which takes as input n data of the same type and produces as output 1 data of the same type. As different expressions can accept the same n data and output the same 1 data, different expressions can be considered the same operation. Thus, an algebra will typically include various **laws** and **identities** that govern the transformation of an arbitrary expression into equivalent expressions. It is particularly desirable for an algebra to be **commutative**, **associative**, and **unital**.

An **algebraic data type(ADT)** is a type that can potentially combine any recursive or non-recursive variant or recursive type, able to be constructed and deconstructed according to an algebra. In accordance with this algebra, variants are also called **sum types**, and composite types are called **product types**.

3.2 Explanations: A programmer's view

In practice, algebraic data types are very simple. The sum type is used to say that data *can* be this *or* that. The product type is used to say that data *must* be this *and* that. The Haskell programming language expresses this neatly:

```
-- General forms:
data MySumType     = MyTypeA | MyTypeB
data MyProductType = MyTypeA MyTypeB

-- Examples:
data Possibly someType = Actually someType | Nothing
data Customer = ID Name ContactInfo PurchaseHistory

-- Example combining sum and product types:
data Tree someDataType = Empty
                       | Branch Tree someDataType Tree
```

In the C programming language, we can accomplish sum types using tagged unions. When a value is assigned, the tag (represented by a `char` in the following example) is set accordingly. Later when the value is retrieved, the tag is used to select the appropriate type:

```
typedef struct MySumType {
  union {
    MyTypeA a;
    MyTypeB b;
  } value;
  char type; // This tag indicates which unioned type is in use
} MySumType;

typedef struct MyProductType {
  MyTypeA a;
  MyTypeB b;
} MyProductType;

/* We can substitute null pointers for
 * tagged unions in the tree example:
 */
typedef struct Tree {
  struct Tree* left;
  struct Tree* right;
  void* data;
} Tree;
```

Dynamically typed languages such as JavaScript naturally have variant and composite types, which can be combined recursively or non-recursively. However, constructing algebraic data types is incredibly laborious in such a language. One might ask, "Why? If I have that much freedom in such a language, wouldn't it be easy?"

Referring to our definition above, an ADT must be "able to be constructed and deconstructed according to an algebra." For an algebra to be effective, the composition of types in a data structure must be consistent. We would need to programmatically enforce strong and static typing in a language that by default supports neither.

3.3 Relevance: The rationality of code

Perhaps the greatest advantage to algebraic data types is how it lends itself to the **rationality** of code. This is not simply how well one can *understand* the code, but how well one can *reason* about it. That is, can you simply look at code and draw reasonable conclusions about it, or must you explore more details of the code base? By way of example, consider the following:

```
char* str = foo();
printf("%d", atoi(str));
```

Can you easily reason about what each function is doing? What does `atoi` do? Can we expect consequences unseen in this code snippet? What is a `char*`? And what is that `"%d"` anyway?

```
String str = foo();
System.out.println(Integer.parse(str));
```

One can easily understand that this code snippet accomplishes the same task as before, but its semantics are different. We still do not know what `foo` does, but beyond that we can easily say that it takes a string, converts it to an integer, and then prints the result.

Similarly, recall the `Tree` examples from the Explanations section above:

```
-- In Haskell:
data Tree someDataType = Empty
                       | Branch Tree someDataType Tree
```

```
// In C:
typedef struct Tree {
  struct Tree* left;
  struct Tree* right;
  void* data;
} Tree;
```

Both are fine examples of defining a tree in their respective languages, both have the same capabilities, and both are easy enough to understand if you examine their parts carefully. But how reasonable is each?

In the first, a Haskell programmer can conclude that a Tree of some type can only be one of two things: empty, or a branch. A branch consists of references to objects with types Tree, some type, and Tree. On further reasoning, we can easily conclude that the first case is a base case, and the second an inductive step, making it clear how this data structure is made finite.

The second is similar; we can see that it consists of a left and right branch, along with data. We may not be sure what `void*` means, but it is some kind of data. It's unclear under what circumstances the tree structure ends; we are expected to understand that `struct Tree*` is a pointer, to remember that pointers can be `null`, and to conclude that `null` likely represents an empty tree.

In a proper algebraic data type, all of the possibilities and edge cases are laid out clearly for the programmer to see and reason about. The data cannot be of a type that is not in its description – in particular, it cannot unexpectedly commute to `null`.

3.4 Relevance: The safety of code

In programs, **safety** is considered to be a combination progress and preservation. **Progress** means that an operation is safe if it results in either other safe operations or the correct termination of the process. **Preservation** means that the state and data of the program remain consistent from one atomic operation to the next.

More specifically, in **type safety** progress means that an expression evaluates to either a value or another expression. Preservation in type safety means that an expression is of the same type as what it evaluates to.[*citation needed*]

Algebraic data types are very strong in terms of both program and type safety. Type safety is guaranteed by definition of its algebra. Let us revisit our definitions: "Any algebraic expression can be considered as an n-ary operation, which takes as input n data of the same type and produces as output 1 data of the same type."

This means that an ADT's type can always be completely evaluated, satisfying the progress principle. It also means that an ADT can be structured in different ways that are equivalent to one another; thus the way data is stored and accessed can morph based on an application's needs, while guaranteeing the preservation of data. In this way, algebraic data types also guarantee program safety, if they are used correctly. (I.e. a programmer can treat two ADTs as equivalent, but it is up to her or him to ensure that they truly are equivalent.)

3.5 Truth functionality and set theory

Algebraic data types have many interesting relations with **truth functionality** (logical functions like `and`, `or`, etc) and **set theory** (union, intersection, etc).

Recall from our definitions that a data type defines, or is defined by, a set of possible values. Therefore, it stands to reason that as we compose data types algebraically, the set of possible values for those data types are also composed via the algebra of sets.

However, we are not just interested in possible values, but actual values as well. An actual value can be described in terms of *truly* being one of the possible values, and *not* being any of the other values. Thus as we compose data types algebraically, the actual values of those data types are also composed via the algebra of booleans.

In product types, therefore, the possible values are described by Cartesian product($\times$), while the actual values are described by logical conjunction ($\wedge$). In sum types, the possible values are described by disjoint union ($\cup$), while the actual values are described by logical disjunction ($\vee$).

3.6 Data Type Algebra

For a transformation of an algebraic data type D to result in an equivalent type, it must be consistent in the corresponding transformations to its possible values P and actual value $A \in P$. In other words, there is a correspondence $A \Rightarrow (D, P)$, and an operation $f : A \to A'$ is such that $A = A'$ if and only if $A' \Rightarrow (D', P')$ where $D = D'$ and $P = P'$.

In terms of the actual values, this is straightforward because our data type algebra's duality is the sum and product, which correspond directly to boolean algebra's standard duality of logical conjunction and disjunction.

Possible values are more difficult, because our algebra there uses a non-standard duality of disjoint union and Cartesian product. (The standard algebra of sets uses union and intersection.) However, we have the advantage that our Cartesian product is non-standard: Cartesian products are not typically commutative, but *is* for commutative for our purposes because the ordering of pairs does not matter in product types.

For our purposes, the Cartesian product is also associative. For example, Triplet1 and Triplet2 are equivalent in the following code snippet:

```haskell
data Pair a b = a b
data Triplet1 a b c = a Pair b c
data Triplet2 a b c = a b c
```

Surprisingly, the Cartesian product is *partially* distributive for our purposes. This is very useful as in some cases we can gain insights into how to compact data, or in the following case, how to express information in a more understandable fashion. For $A \times (B \cup C) = (A \times B) \cup (A \times C)$:

```haskell
data Possible x = Actual x | Nil    -- A(x) + L
data Marriage = Name Possible Name  -- N * (A(N) + L)
-- Distributed version:
data MarriageToSomeone = Name Name  -- N * A(N)
data MarriageToNobody  = Name Nil   -- N * L
data Marriage2 = MarriageToSomeone | MarriageToNobody
                                    -- (N * A(N)) + (N * L)
```

Unfortunately, `Location2` does not hold, and so it is not fully distributive. In the following counterexample, the possible values for `Location` must be either complex latitude-longitude coordinates, or completely unknown. The possible values for `Location2` include the possibility that latitudes may be known while longitudes unknown, or vice-versa. Thus `Location2`'s possible values are a superset of `Location1`'s possible values, and the two ADTs are not equivalent. For $A \cup (B \times C) \neq (A \cup B) \times (A \cup C)$:

```haskell
data Coordinates = Latitude Longitude -- A * O
data Location = Unknown | Coordinates -- U + (A * O)
-- Distributed version:
data MaybeLatitude  = Unknown | Latitude      -- UA + A
data MaybeLongitude = Unknown | Longitude     -- UO + O
data Location2 = MaybeLatitude MaybeLongitude --(UA + A) * (UO + O)
```

Thus, our algebra is dual of sum and product with the following properties:

- **Commutative:**
  $A + B = A + B$
  $A \times B = B \times A$

- **Associative:**
  $A + (B + C) = A + B + C$
  $A \times (B \times C) = A \times B \times C$

- **Distributive sums:**
  $A \times (B + C) = (A \times B) + (A \times C)$
  $A + (B \times C) \neq (A + B) \times (A + C)$

- **Idempotent sums:**
  $A + A = A$
  $A \times A \neq A$

- **Monotonic sums:**
  $A \rightarrow B \Rightarrow (A + C) \rightarrow (B + D)$
  $A \rightarrow B \nRightarrow (A \times C) \rightarrow (B \times D)$

4 Deltas as ADTs by Example

Section incomplete

1. Problem statement: Source code editor
2. Types of source code deltas
3. Defining a generic delta ADT
4. Operational semantics

5 Results

Pending Project 2

1. What did and did not get done
2. Source code repository
3. Metrics - I'm not sure what kind of metrics I'll be able to find with this stuff, but I should put it in this section.

6 Conclusion

Restate the introduction, but this time addressed to someone who has the knowledge your paper offers:

- What you did
- What you discovered
- If you operated from a hypothesis, was it supported?
- Link hard data results to hypothesis
- What was learned?
- Answer any questions explicitly raised in the introduction
- Were objectives achieved?
- Errors and problems encountered?
- Any remaining uncertainties?
- Propose avenues for future exploration, experimentation
- Propose additional questions
- Relate your research to others'
- Final statement

References

[1] Buba, Z. P., & Wajiga, G. M. (2011). *Cryptographic algorithms for secure data communication.* International Journal of Computer Science and Security (IJCSS), 5(2), 227-243.

[2] Hellerstein, D. M., Goland, Y. Y., Feldmann, A., Mogul, J. C., Krishnamurthy, B., Douglis, F., & Hoff, A. V. (2002). *Delta encoding in HTTP.* Delta.

[3] Myers, E. W. (1986). *An O(ND) difference algorithm and its variations.* Algorithmica, 1(1-4), 251-266.

[4] Mogul, J. C., Douglis, F., Feldmann, A., & Krishnamurthy, B. (1997, October). *Potential benefits of delta encoding and data compression for HTTP.* In ACM SIGCOMM Computer Communication Review (Vol. 27, No. 4, pp. 181-194). ACM.

[5] Okasaki, C. (1999). *Purely functional data structures. Cambridge University Press.* Retrieved from http://www.cs.cmu.edu/ rwh/theses/okasaki.pdf

[6] Pierce, B. C. (2002). *Types and programming languages.* MIT press.

[7] van Hoff, A., & Payne, J. (1997). *Generic Diff Format Specification.* Technical Report NOTE-GDIFF, World Wide Web Consortium.