11-1-1994

Incremental Equational Programming

Samuel A. Rebelsky
Dartmouth College

INCREMENTAL EQUATIONAL PROGRAMMING

Samuel A. Rebelsky

Technical Report PCS-TR94-240

11/94

Incremental Equational Programming

Samuel A. Rebelsky

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

Abstract. This paper extends Equational Programming (EP)—a declarative, symbolic programming language—to allow programs to manipulate incrementally defined and modified input terms and to avoid repeated work when evaluating these incremental terms. This paper presents two key aspects of this extension: a notation for representing revision and a modification to EP's runtime library to accommodate this notation.

Unlike Field's method of incremental term rewriting [Fie93], which is designed for more general but less efficient term-rewriting systems, this paper's method accommodates EP's restrictions (in particular, EP's decision to disallow overlapping rules) so that it may take advantage of EP's speed.

1 Introduction

Traditionally, programs manipulate *static* data—once data are specified, entered or computed, those data do not change. Yet data are usually *dynamic* rather than static: expressions change as users realize they mistyped something, databases get new entries, papers change as writers revise text. Often, the only way to handle a change to the input is to completely rerun the program.

If a change to the input is small and should have only a local effect, one would hope that programs could avoid repeated work when they compute the modified output value that corresponds to the modified input value. For example, suppose one ran a list-reversal program on input `[a,b,c]` (yielding `[c,b,a]`). If the `a` in the input list were changed to a `d`, the reverse program might reuse the previous computation so that it takes only one step to convert the `[c,b,a]` to `[c,b,d]`.

While numeric computations appear highly sensitive to modified input, symbolic computations, like the list reversal example, appear to be less sensitive. Hence, it may be possible to extend a symbolic programming language to provide for incremental computation. This paper investigates extensions to Equational Programming (EP) [O'D85]—a lazy, symbolic, declarative programming language—that allow it to compute with incrementally changed terms. The remainder of section 1 introduces key aspects of EP, including restrictions EP places on input programs, EP's I/O system, and certain parts of EP's implementation. Section 2 presents a notation for revision that does not require changes to the syntax or semantics of EP. Given that representation, section 3 suggests possible extensions of EP to automatically accommodate incremental computation. Finally, section 4 uses a small example to illustrate the benefits and drawbacks of this extension.

1.1 Equational Programming

Equational Programming (EP) is a declarative, symbolic programming language in which a program is a set of equations. A compiled EP program lazily transforms an input term (a question) into an equivalent output term (an answer). Current implementations [Str88, SS90, Bai92] treat the equations as directed rewrite rules and compile them to a fast pattern-matching automaton. A compiled EP program repeatedly finds the outermost portion of its input term that corresponds to a left-hand-side, and replaces the portion of the term with the right-hand-side of the equation.

In many ways, EP resembles lazy functional languages, like Haskell [HPW92], but there are key features that distinguish EP from traditional lazy functional languages. Two significant differences relate to the way EP handles symbols and equations. While most functional languages distinguish functors (function symbols that "compute") from constructors (function symbols that build structure), EP does not classify function symbols as functors or constructors. In EP, a symbol can be a constructor in one equation and a functor in another. Functional languages usually prioritize the equations in a program; if both the nth and (n+1)st equations can apply to the same portion of the term, the nth equation is used. EP does not prioritize the equations in a program; the hundredth equation in a program is as likely to be applied as the first.

These differences affect the programs EP accepts as valid. In particular, EP disallows *overlapping* sets of equations: an EP program may not contain two patterns (left hand sides of equations) that apply to the same portion of a term. For example,

```
or(true,X) = true                    (or-1)
or(X,true) = true                    (or-2)
or(false,false) = false              (or-3)
```

is a legal functional program, but not a legal EP program, because both **or-1** and **or-2** apply to the term `or(true,true)`. This difference does not affect the power of EP, but it does force programmers to carefully consider the implications of the programs the write. The functional `or` is strict in its first argument because an equation with a nonvariable in the first argument appears first. To get the same effect, an equational programmer might write

```
or(true,X) = true                    (or-4)
or(false,X) = X                      (or-5)
```

The definition that uses equations **or-4** and **or-5** works as well as the definition that uses **or-1**, **or-2**, and **or-3**. The newer definition also makes clearer that `or` is strict in its first argument.

There are other benefits to EP's restrictions. By restricting programs to be *forward branching* [Str89], implementations are able to produce very fast pattern-matching automata. Using this automaton, the time to find a matching pattern in the term is linear in the size of the pattern matched and not related to the size of the program.

1.2 Implementation of EP

As mentioned above, an EP compiler transforms an input program (a set of equations) into a fast pattern-matching automaton. There are three parts to each compiled EP program:

- The pattern-matching automaton that depends on the input program. The EqStabilize function provides the interface to the pattern-matching automaton. EqStabilize takes a term as input, repeatedly applies the automaton, and returns a root-stable version of the term—a version in which no additional rule applications can affect the root of the term.
- A library of term manipulation utilities that are reused for each program. The most important functions in the term library are GetSymbol and GetChild, which the pattern-matching automaton uses to get symbols and subterms.
- An I/O interface that is reused for each program. This interface controls the operation of the EP program. It builds the input term and calls `EqStabilize` to prepare portions of the term for output.

We return to these functions in section 3.3.

1.3 Lazy Input and Output for Equational Programming

Rather than taking the traditional view that a program's input and output are sequences of data, EP uses an I/O system that corresponds more closely to its model of computation: an EP program reads an input term and generates an output term. This I/O is implicit, not explicit: EP provides no explicit I/O operations.

We have recently extended the I/O system of EP to make this term-I/O lazy [Reb93]. Using this system of I/O, an EP program only generates portions of its output term when a client (e.g., the user) requests those portions and only inspects portions of its input term when it needs those portions to generate the requested portions of the output term. Both input and output terms have a fully-defined upper structure and an undefined fringe.

One can view both the input and output term as having a fringe of *holes* that can be filled in. The program fills in holes in the output term and asks for holes in the input term. The set of equations determines the relationship between the input and output holes.

While this I/O system provides for incremental development of input and output, it does not provide for arbitrary revision; once a hole in the input term is filled in with a particular symbol, that symbol cannot change. Similarly, once an EP program defines a portion of the fringe of its output term, that portion cannot change. This does not prevent further changes to terms as a definition often extends the fringe: function symbols often have additional arguments that are initially defined.

This form of I/O permits a form of incremental computation. Suppose the input to a program is `reverse([?`$_1$`,?`$_2$`,?`$_3$`])`, where the question marks represent holes in the input. The program can, upon demand, determine that the output is `[?,?,?]` without inspecting the

input holes.[1] That is, it can reverse a list without knowing the elements of the list. The user can then alternately request elements of the reversed list and define elements of the input list as the program requests them. This alternation allows the user to see the relation between the input and the output. For example, if the user requests the third element of the output list, the EP program will request the first element of the input list.

This form of I/O also affects the way a user might view input and changes to input. Consider the task of obtaining the first five primes. In a nonincremental system, the input would need to resemble `firstn(5,primes())`. To obtain the first six primes, one would have to rerun the program with input of `firstn(6,primes())`. Given a lazy I/O system, one need only make the input `primes()` and request as much of the output as necessary. To see the first five primes, one asks for the first five elements of the output list. To see the sixth prime, one simply asks for the next element of the output list.

2 Representing Incremental Revision

While the term-based I/O system for EP allows certain kinds of incremental input/evaluation and obviates the need for others, it does not provide a sufficiently general mechanism. Consider input `reverse([a,b,c])`, which one expects to evaluate to `[c,b,a]`. If the user changes the `a` to `d`, then it should take only one step to change the `[c,b,a]` to `[c,b,d]`. If the user appends a `d` to the input list, then the earlier reversal of `[a,b,c]` should contribute to the reversal of `[a,b,c,d]`.

A notation for revision should both accommodate and take advantage of EP's I/O system. It can accommodate EP's I/O system by ensuring that once an output symbol is generated, that output symbol may not change. It can take advantage of EP's lazy I/O by representing all incremental changes to the term by changes at the fringe of the term.

In this new incremental notation, a special Δ symbol marks positions in the term that can change.[2] Δ is a binary symbol; the first argument is the original version of the term at that position, the second argument is the new version of the term. For example, Δ`(a,?)` indicates that `a` might change; Δ`(nil,?)` at the end of a list indicates that elements may be appended to the list.

Using a modified version of the running example, `reverse([`Δ`(a,?),b,c])` evaluates to `[c,b,`Δ`(a,?)]`.[3] Once the input `?` is filled in with `d`, a program may quickly propagate that change and replace the output `?` with `d`. The output term is then `[c,b,`Δ`(a,d)]`.[4] Similarly,

---

[1] The `?`'s on the fringe of the output list do not have indices because the interface does not allow the program to specify the precise relationship beween input and output holes.

[2] Field also uses this notation in [Fie93]. The two notations were developed independently.

[3] Not all of `[c,b,`Δ`(a,?)]` is available for output. In particular, while `c`, `b`, `a`, the list constructors, and the Δ are stable, the `?` is not. Presumably, a special output routine will display this term as `[c,b,a]` and will only attempt to read the `?` when the `a` in the input changes.

[4] Which is displayed as `[c,b,d]`.

`reverse([a,b,c:(Δ([],?))])` evaluates to `Δ([c,b,a],linrev(?,[c,b,a]))`.[5] If one replaces the `?` with `[d]`, the program can generate the new reversed list in two steps, rather than the six it would require to reverse the newly created list from scratch. We return to these examples in section 4.

While the Δ is a simple notion, it needs careful consideration. In particular, a term may have many portions that can change and there should be a way to distinguish the Δ's at those different locations. An annotation attached to each Δ allows one to distinguish Δ's and to relate Δ's in the output term to Δ's in the input term. This annotation appears as a subscript of the Δ, as in Δ_A. When the annotation of a Δ is unimportant, it may be elided.

Since the notation allows multiple Δ's, it is important to understand why and how Δ's nest and the meaning of a term with nested Δ's. Consider the term `f(a)` and suppose that both the whole term and the inner `a` may change. The Δ-term Δ_1`(f(`Δ_2`(a,?`$_2$`),?`$_1$`)` represents these two possibilities. In addition, a portion of a term may change more than once. For example, `a` may become `b`, that `b` may become `c`, and that `c` may change. The Δ-term Δ_1`(a,`Δ_2`(b,`Δ_3`(c,?`$_3$`)))` represents this sequence of changes.

Note that the series of Δ's in the previous example are linked along the second (i.e., *new*) argument. Does it also make sense to have Δ's linked along the first argument, e.g., Δ_1`(`Δ_2`(`Δ_3`(a,?`$_3$`),?`$_2$`),?`$_1$`)`? Yes, such structures make sense. In fact, they get created as the program rewrites Δ-terms. Consider the term Δ_1`(f(`Δ_2`(a,?`$_2$`),?`$_1$`)` and program

```
f(a) = b                                                    (f-1)
```

In order to apply rule **f-1** to that term, one needs to swap the Δ_2 and the `f`. After swapping and applying rule **f-1**, one obtains Δ_1`(`Δ_2`(b,f(?`$_2$`),?`$_1$`)`. The pair of Δ's in this term indicate that the output can change from `b` if the either change 1 or change 2 is made in the input. The order of the Δ's is also important; because Δ_1 is "above" Δ_2, it takes priority. If the change that corresponds to Δ_1 is made to the input, the change that corresponds to Δ_2 can no longer have any effect. A group of Δ's nested along their first argument is called a *Δ-stack*.

2.1 Transforming Δ-Terms into Traditional Terms

Users need not directly manipulate Δ-terms; a special term-builder/term-browser can hide the details of Δ terms from the user. The term builder presents the user with a standard view of the term and converts changes to the term to Δ representation. The browser reads a term with Δ's and, using knowledge of which Δ's in the input term were applied, displays the appropriate portion of the output term using an algorithm like that appears below.

The analysis of the meaning of terms with multiple Δ's suggests an algorithm that extracts the current version of a Δ-term. Note that the meaning of "current" depends on the changes

---

[5]This example uses the linear list reversal program defined in section 4. As above, only part of `Δ([c,b,a],linrev(?,[c,b,a]))` is stable and ready for output. In particular, the outermost Δ and the `[c,b,a]` list are stable; the `linrev` term needs the `?` to be defined before it can be evaluated.

that were made, so the function to extract the current version needs two arguments: a Δ-term and a set of indices that correspond to changes.

```
Term CurrentVersion(Term t, Set Annotations) {
  if IsDeltaTerm(t) then {
    if contains(Annotations,Annotation(t))
      return CurrentVersion(GetChild(t,OLD));
    else
      return CurrentVersion(GetChild(t,NEW));
  }
  else {
    Term new;
    SetSymbol(new,GetSymbol(t));
    for(i = 1; i <= NumChildren(t); ++i)
      SetChild(new,i,CurrentVersion(GetChild(t,i)));
    return new;
  }
}
```

Note that this is a relatively primitive algorithm that only provides one version of an incremental term. [Reb93] describes a more general version.

3 Pattern-Matching with Δ-Terms

Section 2 showed how to annotate terms with Δ symbols to indicate where changes might take place, described ways such annotations might allow programs to reuse work, and developed some interpretations for sequences of Δ's. While programmers could incorporate Δ rules in their programs, a better tactic is to automatically extend EP to handle Δ notations so that programs can match and rewrite annotated terms. There are many options available, including:

- add general rules that depend on the symbols in the program, but not on the patterns in the program,
- add rules that depend on rules in the original program, or
- modify the run-time library.

The first two options are more desirable because they may apply in other contexts. For example, one might be able to use similar techniques modify a functional program to handle revision. Unfortunately, both have drawbacks, which is why we also consider the third option.

3.1 Adding General Rules

The first change one might attempt is to add simple rules that propagate Δ's upward in the program. For each function symbol and for each argument position, *i*, of that function symbol, add a rule of the form

$$\begin{aligned}&\mathrm{Sym}(X_1, X_2, \ldots, X_{i-1}, \Delta(\mathrm{Old}, \mathrm{New}), X_{i+1}, \ldots, X_n) = \\ &\quad \Delta(\mathrm{Sym}(X_1, \ldots, X_{i-1}, \mathrm{Old}, X_{i+1}, \ldots, X_n), \\ &\quad\quad \mathrm{Sym}(X_1, \ldots, X_{i-1}, \mathrm{New}, X_{i+1}, \ldots, X_n))\end{aligned} \qquad \textbf{(ΔSym-i)}$$

This rule switches the position of the function symbol and a Δ in the *i*th position.[6] For example, given program

```
f(a) = b                                                    (f-1)
```

this strategy adds the rule

```
f(Δ(Old,New)) = Δ(f(Old),f(New))                            (Δf-1)
```

Given input of `f(Δ(a,?))`, the new **Δf-1** rule applies and the term rewrites to `Δ(f(a),f(?))`. Rule **f-1** then applies and the term rewrites to `Δ(b,f(?))`. Only the Δ and the `b` in this term are available for output; when the `?` is filled in, further rules may apply.

There are two drawbacks to this strategy. One is that these new rules can propagate Δ's unnecessarily. Consider the program

```
g(a,X) = b                                                  (g-1)
```

The strategy of adding one rule per position means adds two rules:

```
g(Δ(Old,New),X) = Δ(g(Old,X),g(New,X))                      (Δg-1)
g(X,Δ(Old,New)) = Δ(g(X,Old),g(X,New))                      (Δg-2)
```

given input of `g(a,Δ(a,b))`, one can apply rule **Δg-2** and rewrite the term to `Δ(g(a,a),g(a,b))`. Rule **g-1** applies to both subterms, which become `b`. The final term is `Δ(b,b)`. If rule **g-1** had been applied first, the final term would have been `b`.

This example illustrates a second drawback to this strategy: it creates overlapping sets of rules. Both **g-1** and **Δg-2** apply to the term `g(a,Δ(a,b))` and both **Δg-1** and **Δg-2** apply to `g(Δ(?,?),Δ(?,?))`. Since EP disallows overlapping sets of rules, this strategy fails.

3.2 Adding Modified Rules

The previous method failed, in part, because it propagated Δ's even where there were variables in the original term,. Hence one might instead base the Δ-rules more closely on the original rules: for each rule in the original program and for every combination of nonroot nonvariable positions in that rule, add a rule with a Δ and the corresponding nonvariable in each of those positions. For example, given rules,

```
f(a) = b                                                    (f-1)
g(b,b) = a                                                  (g-2)
```

add the rules[7]

```
f(Δ(a,New)) = Δ(f(a),f(New))                                (Δf-2)
g(Δ(b,New),b) = Δ(g(b,b),g(New,b))                          (Δg-3)
g(b,Δ(b,New)) = Δ(g(b,b),g(b,New))                          (Δg-4)
g(ΔA1(b,New1),ΔA2(b,New2)) =                                (Δg-5)
```

---

[6] Field suggests adding a similar set of rules in [Fie93].

[7] The right-hand-sides of these rules can be made somewhat more efficient by replacing `f(a)` with `b` in rule **Δf-2** and `g(b,b)` with `a` in rules **Δg-3**, **Δg-4**, and **Δg-5**. In this example, the less efficient versions are used to clarify the purpose of the rules.

Δ_{A1}(g(b,b),Δ_{A2}(g(b,New2),
Δ_{A2}(g(New1,b),g(New1,New2)))

Note that the right hand side of equation **Δg-5** is rather complex. This is because the new rule needs to allow both the change sequence in which the change that corresponds to Δ_{A1} precedes the change that corresponds to Δ_{A2} and the sequence in which Δ_{A2} precedes Δ_{A1}. Given the choice to put Δ_{A1} at the root of the term, both subterms must then have a Δ_{A2} at the root. These right-hand-sides become hard to read, especially as they get larger. This is not the only problem with this strategy.

The biggest problem with this strategy is that it creates so many rules. If there are n nonroot nonvariable positions in the original rule, then this strategy adds 2^n-1 new rules. While the number of rules does not affect the time to match a pattern (one of the benefits of using EP's pattern-matching strategy), it does significantly affect the size of the pattern-matching automaton and the time it takes to compile the program.

Even this might be acceptable if the strategy worked. Unfortunately, it does not. In particular, it does not generate rules that match a term that has a Δ-stack in one of the nonvariable positions. In the example above, none of the newly added rules match the term g(Δ_{A1}(Δ_{A2}(a,?),?),a). The only way to be able to match this type of term would be to add rules that allow a size-two Δ-stack at every position. But these would still fail to match input terms with size-three Δ-stacks. No matter how many rules this strategy adds, there are still terms that can't be matched.

While one might hope to alleviate these problems by using variables for the first argument to the Δ's in these patterns, such rules may also overlap. These further problems are discussed in chapter 9 of [Reb93].

3.3 Modifying the Run-Time Library

The problems encountered trying to add new rules that accommodate Δ's call for a different approach. Rather than adding rules, one might make small changes to EP's runtime library so that EP programs can match Δ-terms. One strategy is to propagate Δ's upwards (i.e., swap a Δ with a parent) only when that swap may help EP match one of the patterns in the original program.

Recall that a compiled EP program provides an `EqStabilize` routine that, given a term as input, generates the root-stable version of that term. Since Δ's do not appear in the original program, any term with a Δ in a significant position (i.e., one that the pattern-matching automaton inspects) will immediately be marked as root stable. By keeping track of the positions the automaton inspects and the order it inspects those positions, a new stabilize routine can determine whether a Δ needs to be propagated upwards.

This solution requires modifications to the `GetSymbol` and `GetChild` functions, and a more general stabilization routine that calls `EqStabilize`. A stack keeps track of the edges the automaton traverses. Another variable keeps track of the last node visited. The new stabilize routine, `MainStabilize`, depends upon `EqStabilize` to do most of the work. If the last

subterm that `EqStabilize` inspects has a Δ at the root, `MainStabilize` determines the path to that node and propagates the Δ upward to the root of the original term. It then returns the new term with the Δ at the root (since that term is now root stable).

```
Term MainStabilize(Term t)
{
   push(END);
   Term stab = EqStabilize(t);
   // Propagate the Δ upwards
   if (GetSymbol(lastVisited) = DELTA) {
      path = GetPath(stab,lastVisited);
      stab = PropagateChange(path,lastVisited);
   }
   // Clear anything pushed on the stack
   while (pop() ≠ END)
     ;
   return stab;
}
```

In this code, `GetPath` uses the pushed edges to find the path from stab to lastVisited. `PropagateChange` propagates a Δ up through a path in the term. `GetSymbol` and `GetChild` require only minor modifications.

```
Symbol GetSymbol(Term t)
{
   // New code
   lastVisited = t;
   // remainder as before
   …
}

Term GetChild(Term t, Integer childNum)
{
   // initial part as before
   …
   // New code
   push(Triplet(t,childNum,child));
   return child;
}
```

Because any Δ encountered by the pattern-matching automaton is propagated to the root of the term that was being matched, this updated run-time library allows EP programs to match and rewrite Δ-terms. The update to an experimental version of the EP library took less than two pages of code and did not significantly affect the running time of EP on nonincremental inputs.

This is not to say that there are not drawbacks to this method of matching Δ-terms. In particular, it can be expensive. Suppose a pattern of size n would match the term if the Δ's were not there; it can take $O(n^2)$ time to propagate all the Δ's high enough that the pattern matches. If one of the earlier methods (such as the one in section 3.2) had worked, then it would only take $O(n)$ time to match. This is not a severe problem, as most EP patterns are small.

4 An Example

Let us see what happens when the above technique is used to reverse a list completely annotated with Δ's. This example uses linear reversal program given by the following equations:

```
reverse(L) = linrev(L,nil)                                  (rev-1)
linrev(nil,R) = R                                           (lr-1)
linrev(cons(H,T),R) = linrev(T,cons(H,R))                   (lr-2)
```

The steps in reversing `[a,b]` are:

```
reverse(cons(a,cons(b,nil)))                                [rev-1]
= linrev(cons(a,cons(b,nil)), nil)                          [lr-1]
= linrev(cons(b,nil), cons(a,nil))                          [lr-1]
= linrev(nil, cons(b,cons(a,nil))                           [lr-2]
= cons(b,cons(a,nil))
```

There are five symbols/positions in `[a,b]`. If each of these is extended with a Δ in `reverse[a,b]` and the modified EP is run on that term, the program executes the following sequence of reductions and Δ-swaps.

```
reverse(Δ1(cons(Δ2(a,?2),                                   [rev-1]
                Δ3(cons(Δ4(b,?4),Δ5(nil,?5)),?3),?1)
= linrev(Δ1(cons(Δ2(a,?2),                                  [Δ-swap]
                 Δ3(cons(Δ4(b,?4),Δ5(nil,?5)),?3),?1),
         nil)
= Δ1(linrev(cons(Δ2(...),Δ3(...)),nil),linrev(?1,nil))      [lr-1]
= Δ1(linrev(Δ3(cons(...),?3),[Δ2(...)]),linrev(?1,nil))     [Δ-swap]
= Δ1(Δ3(linrev(cons(Δ4(...),Δ5(...)), [Δ2(...)]),           [lr-1]
        linrev(?3,[Δ2(...)]),
    linrev(?1,nil))
= Δ1(Δ3(linrev(Δ5(nil,?5),[Δ4(...),Δ2(...)]),               [Δ-swap]
        linrev(?3,[Δ2(...)]),
    linrev(?1,nil))
= Δ1(Δ3(Δ5(linrev(nil,[Δ4(...),Δ2(...)]),                   [lr-2]
           linrev(?5,[Δ4(...),Δ2(...)])),
        linrev(?3,[Δ2(...)]),
    linrev(?1,nil))
= Δ1(Δ3(Δ5([Δ4(b,?4),Δ2(a,?2)]),                           [done]
           linrev(?5,[Δ4(...),Δ2(...)])),
        linrev(?3,[Δ2(...)]),
    linrev(?1,nil))
```

This term cannot be further evaluated until the `?`'s are filled in. It took four traditional rewrite steps and three Δ propagation steps to reach this point. Evaluating `reverse([a,b])` took four traditional rewrite steps. Hence, the only added cost was in the Δ-propagation. If the `a` were changed to `c`, (change Δ_3, then the new reversed list can be read from the Δ term. If `c`

were appended to the original list (a change from `nil` to `[c]` at Δ_5),[8] the program can then execute the following steps

```
linrev(?,[Δ4(...),Δ2(...)])                    [add [c]]
= linrev([c],[Δ4(...),Δ2(...)])                [lr-1]
= linrev(nil,[c,Δ4(...),Δ2(...)])              [lr-2]
= [c,Δ4(b,?4),Δ2(a,?2)]                        [done]
```

Using only two more steps, the program has produced the reversal of the extended list. Reversing the new list "from scratch" would take five steps. If the original list were of length n, it would still take only two additional steps to reverse the list with an appended character, while it would take $n+2$ steps to reverse it from scratch.

The Δ notation has clearly reduced the number of rewriting steps, but at some cost. If it were not likely or possible that the term would change at every location, one could save a great deal of effort by only placing Δ's where they're needed. In the list reversal example, Δ's might only be associated with the `nil` at the end of the list and with each element of the list. Then it would only take one extra step to propagate a Δ when reversing the list (for the Δ at the end of the list).

5 Related Work

No discussion of incremental updates to terms would be complete without a mention of Reps' work on efficient incremental updates to attributes in parse trees [Reps84]. Nonetheless, Reps' work focuses on the propagation of changes in a static term, rather than on rewriting changing terms.

Field has recently given an in-depth analysis of a structure similar to Δ-terms for incremental updates in term-rewriting systems [Fie93]. His incremental rewriting relies on rules like those suggested in section 2.1. He is able to use such rules because he works in a system that allows overlapping rules, while EP disallows such rules and therefore calls for a different solution. Field is also able to take additional advantage of overlap by writing Δ-free rules that perform extra work in certain circumstances. Field also observes that it is overly costly to place Δ's in every position in the term.

6 Summary

This paper described Δ-terms, a simple notation for representing incrementally changing terms. Δ-terms encode changes anywhere in a term to changes at the fringe of the term. Because Equational Programs can read and write incrementally extended terms, this notation can serve as the basis of incremental equational programming.

---

[8] Or, more likely, to Δ_6`([c],?)`.

Given Δ notation, it is a nontrivial problem to automatically add Δ-based rules to a traditional equational program. A modification to EP's run-time library provides a simpler extension that accommodates Δ notation. Fortunately, this modification is relatively minor, only affects performance when Δ-nodes are used, and provides relatively efficient incremental computation. An experimental version of EP that incorporates this extension has been successfully used with simple programs.

There are both benefits and drawbacks to the solution based on modifying the runtime library. While some programs run faster, a significant amount of effort can be expended moving Δ's throughout the term. Because of this, we suggest that Δ's be used sparingly. More work must be done on a precise analysis (both theoretical and experimental) of the costs and benefits of this strategy.

Bibliography

[Bai92] Stephen W. Bailey. *Ielp User Guide.* University of Chicago, February 1992. Part of the Ielp distribution available via anonymous ftp from cs.uchicago.edu.

[Fie93] John Field. "A Graph Reduction Approach to Incremental Term Rewriting (Preliminary Report)." In *Proceedings of the 5th International Conference on Rewriting Techniques and Applications,* Springer-Verlag, 1993.

[HPW92] Paul Hudak, Simon Peyton Jones, and Phil Wadler (editors). "Report on the Programming Language Haskell, A Non-Strict, Purely Functional Language, Version 1.2." *ACM SIGPLAN Notices*, **27** (5), May 1992.

[O'D85] Michael J. O'Donnell. *Equational Logic as a Programming Language,* The MIT Press, 1985.

[Reb93] Samuel A. Rebelsky. *Tours, A System for Lazy Term-Based Communication.* PhD thesis, University of Chicago, Chicago, Illinois. Technical report 93-06, University of Chicago Department of Computer Science, 1993.

[Reps84] Thomas W. Reps. *Generating Language-Based Environments,* The MIT Press, 1984.

[SS90] David J. Sherman and Robert I. Strandh. "An Abstract Machine for Efficient Implementation of Term Rewriting." Technical Report 90-013, University of Chicago Department of Computer Science, 1990.

[Str88] Robert I. Strandh. *Compiling Equational Programs into Efficient Machine Code*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1988.

[Str89] Robert I. Strandh. "Classes of Equational Programs that Compile into Efficient Machine Code." In N. Dershowitz (editor), *Proceedings of the 34d International Conference on Rewriting Techniques and Applications*, Springer-Verlag, 1989.