A Differentiation Primitive for Extended λ-Calculus

Terry Flaherty

Department of Computer Information Systems Applications
City College, Loyola University, New Orleans, LA 70118

Abstract

A symbolic differentiation functional that handles expressions containing free and bound variables in an extended λ-calculus programming language is described. The differentiation primitive is implemented by augmenting the set of graph-reduction rules that define the evaluation of expressions. A formalization of partial derivatives of functions wrt position of parameters is presented. A comparison is made to other methods of automatic differentiation.

1. Introduction

Although analytic differentiation was one of the first computer applications, the subject of automatic differentiation is not a closed book. Rall (1981) advises "that a powerful computational tool can be fashioned without excessive effort." Shearer and Wolfe (1985) describe a library of imperative language procedures that includes analytic differentiation. The present work demonstrates that such a widely useful symbolic operator can be added to a functional language, which is more amenable to symbolic manipulation. Moreover, we treat the problem of differentiating expressions that may contain bound variables (within λ-abstractions) as well as free variables.

Previous work in automatic differentiation has been implemented in imperative languages. We show that it is feasible to implement differentiation in the hardware of a λ-calculus graph-reduction machine. The differentiation primitive is defined by a small set of graph-reduction rules to be implemented in hardware along with the standard evaluation rules for such a machine. We will show that the derivative primitive acts as a true functional, operating on functions to produce functions, and complex chain differentiation is performed. We hope that the formalization will shed some light on the theoretical aspects of symbolic differentiation.

The paper is organized into 6 sections. Section 2 discusses automatic differentiation. Section 3 discusses the implications of automatic differentiation within an extended λ-calculus —differentiation as a functional, the symbolic-numeric interface, and the uniqueness of λ-expressions. Section 4 describes the extended λ-calculus language SRL, its syntax and rules of evaluation. Section 5 includes a formalization of differentiable expressions, their derivatives, and a set of graph-reduction rules implementing the derivative primitives. Section 6 offers some concluding remarks.

2. Automatic Differentiation

We use the term automatic differentiation (in the sense of automatic programming) to refer to techniques whose primary goal is the numerical evaluation of the derivative of a function, but which first produce an executable representation of the derivative to obtain the numeric results. The following characteristics distinguish automatic differentiation from general symbolic manipulation: algebraic simplification is not an issue, and its implementation must be relatively simple (in order to compete with general algebraic manipulation systems). Brown and Hearn (1979) discuss the importance of such special-purpose algebraic manipulation.

Rall (1983) gives the theory and describes software which differentiates formulas within a FORTRAN environment.

© 1988 ACM 0-89791-260-8/88/0002/0009 $1.50

3. Differentiation In λ-Calculus

Differentiation as a True Functional

Although differentiation has become a typical exercise in introductory Lisp courses, and huge systems like MACSYMA and REDUCE perform far more complex algebraic manipulations, questions of practical and theoretical interest remain. Can we easily provide the widely useful differentiation operation without the expense of a general algebraic manipulation system? How do we represent differentiation as a functional, especially in systems that treat functions as first-class citizens and hence proper arguments to such a functional? MACSYMA's lack of a true derivative functional has been noted. (Wester and Steinberg 1983, Golden 1985)

Symbolic-Numeric Interface

One difficulty in performing algebraic manipulation is the disparity between the mathematical statement of an input problem and the hardware of the computer. (Barton and Fitch 1972) For example, although we may use the concept of variable in FORTRAN to form the expression "x + x", its meaning within FORTRAN is not the same as the mathematical meaning of the string of symbols. Functional programming languages constructs come closer to the mathematical statement of a problem.

Backus (1978) showed that programs based on function application and combining forms are more easily understood and manipulated than imperative programs, which have fewer useful mathematical properties. Advances in parallel architecture research are making functional programming languages a more viable alternative to imperative languages. Landin (1964) and Turner (1979) describe machines that execute code generated from programs written in a λ-calculus-based language. Revesz (1984, 1985) proposes a graph-reduction technique for evaluating extended λ–expressions directly.

In a λ-calculus reduction machine, the distinction between symbolic computation and numeric computation is blurred. However in an imperative language such as FORTRAN, it is impossible to compile expressions during the execution of a program. Hence a communication problem exists between symbolic and numeric techniques. (van Hulzen and Calmet 1983)

Using λ–notation as our basic representation of a function, we preserve the mathematical meaning of the symbols. Symbols — variables, built-in functions, and numbers — may be manipulated to produce both numeric values as well as symbolic values. If we represent the λ-expression as a graph, then we can apply graph reduction rules to transform the expression into equivalent forms. Some of these forms are "symbolic" in the sense that it contains variables.

Performing differentiation by hardware-implemented reduction rules operating on λ-calculus expressions suggest a viable approach to the symbolic-numeric interface problem. At the same time, we have the advantage of the unambiguous denotation of functions that λ-notation affords.

Expressions Unique to λ-Calculus

In contrast to a FORTRAN numeric expression, a λ-expression may contain a λ-abstraction, which represents a user-defined function.

Abstractions can occur also when a complex formula is expressed in terms of sub-expressions. For example, the set of equations

```
u = y +x*z
v = (x*u)/y
g = x**2 + EXP(v) +u
```

can be expressed in λ-calculus as

λu.(λv.x**2 + EXP(v) + u)(x*u)/y) y+x*z

The abstractions specify the substitution of the sub-expressions into the formula. Substituting (x*u)/y for v in x**2 + EXP(v) +u is equivalent to applying the function λv.x**2 + EXP(v) + u to the argument (x*u)/y.

λu.(λv.x**2 + EXP(v) + u)(x*u)/y) y+x*z denotes the expression g as a function of x.

If the chain rule is viewed as a symbolic transformation, the symbolic constituents are functions and the symbolic constructions are function composition. In a functional language we are freer to choose how to represent functions. However in a non-functional language such as FORTRAN, we need extra apparatus to record dependencies among variables to handle partial differentiation. In the sequel, we show that λ-expressions allow a more complete application of the chain rule and we offer a formalization of partial derivatives wrt position of parameters.

4. Simple Reduction Language (SRL)

The way a bound variable is associated with a value distinguishes implementations of λ-calculus-based languages. In Landin's (1964) SECD machine and the standard LISP interpreter, (McCarthy 1960) the environment is represented with a list of name-value pairs. Turner's (1979) technique eliminates all variables by converting the λ-expression to its pure combinator equivalent. and applies reduction rules to subcomponents of the graph representation of the expression.

Whenever a sub-structure of the graph matches the LHS of a rule, then we replace that sub-structure with the RHS and continue until no rules apply. We have implemented differentiation within Revesz's (1984) Simple Reduction Language (SRL) system which reduces λ-expressions directly. The expression is not converted to combinators and no separate environment structures are required to simulate β-reduction. (Perrot 1979)

SRL extends pure λ–calculus with the inclusion of lists as expressions and a set of built-in functions for operating on lists and integers. Its syntax is given in Table 1.

SRL primitives include arithmetic operations, list functions, the conditional function, and the Y-combinator.

If the graph-reduction rules (see Table 2) are implemented in hardware, then SRL is a human-readable machine language with programs that correspond directly to the structure being manipulated.

Table 1. The Syntax of SRL

```
<λ–expression> ::=<variable> |
          <constant> |
          <list> |
          ( <expression> ) <expression>|
          λ <variable> . <expression>

<list> ::= [] | [ <expression> <list-tail>

<list-tail> ::= ] | , <expression> <list-tail>

<constant> ::=
     <integer> | + | - | * | / |^ | ~ | & | ?
```

A non-atomic expression is either a list , e.g., [x, y, 8], an application, e.g., ((+)7)8, or an abstraction, e.g., λx.λy.((+)x)y. In the application (p)q, p is the operator that is being applied to the operand q. Note that parentheses are required around the operator rather than the operand. In the abstraction

λx.λy.((+)((*)x)x)((*)x)x,

x and y are bound variables, and ((+)((*)x)x)((*)x)x is the body of the abstraction. N-ary functions are curried. For example the sum of x and y is written ((+)x)y.

& is the list construction operator and ^ and ~ denote the head and tail functions. ? denotes the Y-combinator. Built-in list operations include append and inner (which forms the inner product of two given lists of numeric expressions.)

Table 2. SRL Reduction Rules

<u>α-rules: Renaming</u>:

(α-1) $\{z/x\}E \rightarrow z$, if $E = x$

(α-2) $\{z/x\}E \rightarrow E$ if x not in free(E)

(α-3) $\{z/x\}\lambda y.E \rightarrow \lambda y.\{z/x\}E$, if $x \neq y \neq z$ and z is not bound in E

(α-4) $\{z/x\}(E_1)E_2 \rightarrow (\{z/x\}E_1)\{z/x\}E_2$

(α-5) $\{z/x\}[E_1,\ldots,E_n] \rightarrow [\{z/x\}E_1,\ldots,\{z/x\}E_n]$

<u>β-rules: Substitution</u>:

(β-1) $(\lambda x.x)E \rightarrow E$

(β-2) $(\lambda x.E_1)E_2 \rightarrow E_1$ if x not in free(E_1)

(β-3) $(\lambda x.\lambda y.E_1)E_2 \rightarrow \lambda z.(\lambda x.\{z/x\}E_1)E_2$ where z is a new variable not occurring in $(\lambda x.\lambda y.E_1)E_2$

(β-4) $(\lambda x.(E_1)E_2)E_3 \rightarrow ((\lambda x.E_1)E_3)(\lambda x.E_2)E_3$

<u>γ-rules: List Manipulation</u>

(γ-1) $([E_1,\ldots,E_n)F \rightarrow [(E_1)F,\ldots(E_n)F]$

(γ-2) $\lambda x.[E_1,\ldots,E_n] \rightarrow [\lambda x.E_1,\ldots,\lambda x.E_n]$

The graph-reducer reduces the leftmost application first. If the argument to a built-in function requires reduction, then we perform one step in its reduction and try again to apply the function. The argument is reduced only enough to satisfy the conditions of the function.

5. Differentiation Primitives

What kinds of SRL expressions can be differentiated? We want to handle any n-ary function whose body is a composition of both arithmetic primitives and user-defined functions (abstractions).

Def. The class of ***differentiable functions*** are of the form

$\lambda x_1\ldots\lambda x_n.P$,

where the body, P, is a ***differentiable form***.

Def. A ***differentiable form*** is defined recursively as follows.

1. P, where P is a variable or number.

2. (F)P, where F is one of the functions, sin, cos, atan, log, or exp, and P is a ***differentiable form***.

3. ((F)P)Q, where F is one of the functions +, -, *, /, or expt, and P and Q are ***differentiable forms***.

4. $(\ldots(\lambda x_1.\ \ldots\ \lambda x_n.P)a_1)\ldots)a_n$, where $\lambda x_1.\ \ldots\ \lambda x_n.P$ is a ***differentiable function***, and $a_1,\ldots,a_n$ are ***differentiable forms*** .

Def. The ***derivative of a differentiable form***, P, wrt the variable x, denoted by ((diff)x)P, is the formal partial derivative of the fully-evaluated expression P, wrt the free variable x.

Def. The ***derivative of a differentiable function***, of the form $\lambda x_1\ldots\lambda x_n.P$, denoted by (difff) $\lambda x_1\ldots\lambda x_n.P$, is the list

$[\lambda x_1\ldots.\lambda x_n.((\text{diff})x_1)P,\ldots,$
$\lambda x_1\ldots.\lambda x_n.((\text{diff})x_n)P]$.

Differentiation is often cited as a classic example of a functional. However, in the usual approach to symbolic differentiation the functional nature of the process is only implicit. A formula implicitly represents a function of its indeterminate symbols.

We can easily make the functions explicit, using λ–notation, by abstracting on all the variables in the formulas. Thus the derivative of $\lambda x.x^2$ is $\lambda x.2*x$.

This distinction may seem pedantic. However, not only notational purity is at issue. In the application of a user-defined function, then the derivative of this **function** is needed in order to apply the chain rule. We need a derivative primitive that is a functional.

We treat built-in functions and user-defined functions uniformly. Instead of including particular graph-reduction rules for each built-in function, we apply the chain rule and retrieve the derivatives of built-in functions. Thus, $(x^2)*(\sin x)$ is treated as the binary function, *, applied to x^2 and sin x. The list of partial derivatives for the function * is stored essentially as [λx.λy.y, λx.λy.x].

We extend the usual manipulation of algebraic expressions to produce their derivatives to the manipulation of abstractions to produce their derivative functions. Hence we differentiate expressions containing β-redexes.

The derivative of $\lambda x_1. \dots \lambda x_n.P$, which has more than one λ–binding, is a list of partial derivatives. We make use of SRL's ability to handle lists directly to produce the list of partial derivatives wrt each bound variable.

The γ-rules of SRL control the application of lists of curried functions. (([λx.λy.((diff)x)P,λx.λy.((diff)y)P])Q)R represents the application of a list of partial derivatives to the arguments Q and R. The result is the list of applications,

[λx.λy.((diff)x)P)Q)R,((λx.λy.((diff)y)P) Q)R]

Table 3. Reduction Rules for *Diff*

1-a. ((diff)x)(P)Q → ((inner)(((diffb)x)P)Q)((diffa)x)(P)Q, if (P)Q is a ***differentiable form***
1-b. ((diff)x)λy.P → λy.((diff)x)P
1-c. .((diff)x)x → 1
1-d. ((diff)x)y → 0, if y is a variable ≠ x or y is a number

Table 4. Reduction Rules for *Diffb*

2-a. ((diffb)x)(P)Q → (((diffb)x)P)Q
2-b. ((diffb)x)λy.P → ((difff)λx.λy.P)x, if x in free(λy.P) ((diffb)x)λy.P → ((difff)λz.λy.P)z, if x not in free(λy.P) (z is a "fresh" system variable)
2-c. ((diffb)x)f → ((difff)f)x, if f is a built-in function

Table 5. Reduction Rules for *Difff*

3-a. (difff)(P)Q →[], if (P)Q is a ***differentiable form***
3-b. (difff)λx.P →((&) λx.((diff)x)P) λx.(difff)P
3-c. (difff)f, see Table 7, if f is a built-in function

Table 6. Reduction Rules for *Diffa*

4-a. ((diffa)x)(P)Q → ((append) ((diffa)x)P) [((diff)x)Q], if (P)Q is a ***differentiable form***
4-b. ((diffa)x)P →[1], if P is not an application

Table 7. Built-in Function Derivatives

(difff)+ →	[λz.λx.λy.0, λz.λx.λy.1,λz.λx.λy.1]
(difff)- →	[λz.λx.λy.0, λz.λx.λy.1, λz.λx.λy.((-)0)1]
(difff)* →	[λz.λx.λy.0, λz.λx.λy.y, λz.λx.λy.x]
(difff)/ →	[λz.λx.λy.0,λz.λx.λy.((/)1)y, λz.λx.λy.((-)0)((/)x)((*)x)x]
(difff)expt →	[λz.λx.λy.0, λz.λx.λy.((*)y)((expt)x)((-)y)1, λz.λx.λy.((*)((expt)x)y)(log)x]
(difff)sin	→[λz.λx.0, λz.λx.(cos)x]
(difff)cos	→[λz.λx.0, λz.λx.((-)0)(sin)x
(difff)atan →	[λz.λx.0,λz.λx.((/)1)((+)1)((*)x)x]
(difff)log	→[λz.λx.0, λz.λx.((/)1)x]
(difff)exp	→[λz.λx.0, λz.λx.(exp)x]

In the following discussion, the references are to the differentiation reduction rules in Tables 3-6. where x is the variable of differentiation.

Diff Handles Differentiable Forms

Applying diff to an application, we use the chain rule. The derivative is the inner product of the list of partial derivatives applied to the arguments, and the list of derivatives wrt x of each argument. This is sufficient if the body of the operator part of P does not contain free occurrences of x. To handle the case where the body does contain occurrences of x, we add the derivative wrt x of the body of the operator part of P, evaluated at is arguments. (Rule 1-a)

Applying diff to an abstraction results in moving diff inside and applying it to the body . The result is an abstraction with the same bound variables. We reduce ((diff)x)P next. (Rule 1-b)

Rules 1-c and 1-d handle the base cases of differentiation of a differentiable form.

Diffb Handles Body of an Abstraction

Diffb 's purpose is to differentiate an abstraction, producing **both** the partial derivatives of the function as well as the **derivative of the body** of the function wrt the variable of differentiation.

The derivative of the body is obtained by further abstracting the function using the variable of differentiation. For example ((diffb)x)λy.P becomes ((difff)λx.λy.P)x. (When ***difff*** is applied the derivative of the body of the abstraction wrt x is computed.) If λx occurs in the prefix of λy.P, then x does not occur free in λy.P and we must not duplicate our differentiation of P wrt x. In this case, we abstract using a new system variable that is guaranteed not to occur in λy.P. We have ((diffb)x)λy.P → ((difff)λz.λy.P)z. (Rule 2-b)

If ***diffb***'s argument is a built-in function, then ***diffb***'s result is the list of partial derivatives returned by ***difff*** applied to the variable of differentiation (Rule 2-c).

If ***diffb***'s argument is an application, then ***diffb***'s result is the list of partial derivatives of the left-most function of the application, applied to its arguments Rule 2-a insures that diffb moves inside nested applications until it encounters a function. For example

((diffb)x)(((λp.λq.λr.F)x)y)z

reduces after three applications of Rule 2-a to

((((diffb)x)λp.λq.λr.F)x)y)z

and then after several steps to

((([λx. λp.λq.λr.((diff)x)F,

λx. λp.λq.λr.((diff)p)F,

λx. λp.λq.λr.((diff)q)F,

λx. λp.λq.λr. ((diff)r)F])x)y)z.

The head of the list is the derivative of F wrt x. The rest of the list contains the partial derivatives wrt the bound variables of λp.λq.λr.F. This is applied in a curried fashion to the arguments x, y, and z. using the γ-1 rule of SRL.

Diff Handles Differentiable Functions

Difff requires an abstraction or a built-in function for an argument. If the argument is an abstraction, then ***difff***'s result is the list of partial derivatives wrt each bound variable (Rule 3-a). The head of the list is the partial derivative wrt the first bound variable and the tail is the list of partial derivatives wrt the remaining bound variables. Thus the list of partial derivatives is constructed in a telescoping fashion using the γ-2 rule of SRL. For example,

```
(difff)λx.λy.P →
((&)λx.((diff)x)λy.P)λx.(difff)λy.P →
((&)λx.λy.((diff)x)P)λx.(difff)λy.P →
((&)λx.λy.((diff)x)P)
              λx.((&)λy.((diff)y)P)(difff)P→
((&)λx.λy.((diff)x)P)
              λx.((&)λy.((diff)y)P)[]→
((&)λx.λy.((diff)x)P)λx.[λy.((diff)y)P)]
→
((&)λx.λy.((diff)x)P)[λx.λy.((diff)y)P)]
→
[λx.λy.((diff)x)P),λx.λy.((diff)y)P)]
```

The value of difff given a built-in function is the appropriate list of partial derivatives, the first of which corresponds to the derivative of the body of an abstraction.

Diffa Handles Arguments to a Function

Diffa returns the list consisting of 1, followed by the derivatives wrt x of each argument of the function application. In the expression (P)Q, Q is one argument to the function that is contained in P. We place its derivative at the end of the list of derivatives. P contains the other arguments (if present) as well as the function. We apply *diffa* recursively to handle these arguments (Rule 4-a).

Since ***diffa*** is applied to a ***differentiable form***, (...(F)a_1)...)a_n, we will have processed all the arguments a_1, ...,a_n exactly when we reach the operator F (Rule 4-b).

Example . Here are some of the stages in the reduction of the expression (difff)λx.((*)x)x.

```
(difff)λx.((*)x)x
→ ((&) λx.((diff)x)((*)x)x)
λx.(difff)((*)x)x  → [λx.((diff)x)((*)x)x]
→ [λx.((inner) (((diffb)x)(*)x)x)
       ((diffa)x)((*)x)x]
→ [λx.((inner) ((((diffb)x)*)x)x)x)
       ((diffa)x)((*)x)x]
→ [λx.((inner) ((((difff)*)x)x)x)x)
       ((diffa)x)((*)x)x]
→ [λx.((inner) ((([ λz.λx.λy.0,
       λz.λx.λy.y,
       λz.λx.λy.x])x)x)x)x)
       ((diffa)x)((*)x)x]
→ [ λx.((inner) [0, x, x]) ((diffa)x)((*)x)x]
→ [λx.((inner) [0, x, x])
       [((diff)x)x,((diff)x)x]]
→ [λx.((inner) [0, x, x]) [1, 1, 1] ]
→ [λx.((+)x)x]
```

6. Concluding Remarks

Comparison to Other Projects

In SUPER-CODEX, each formula is compiled into a code list and its name (LHS) is entered into a table. (Rall 1981) Whenever a function code list is differentiated, the resulting code list is entered into the table. Differentiating a formula f wrt a variable x also places f into a list of variables that depend on x. These "dependency tables" are used to determine whether a variable occurrence is treated as a constant or whether its derivative code list exists and can be used. Rall reports that the most time-consuming segment of SUPER-CODEX is the search of the dependency table.

Perhaps the greatest recommendation for incorporating differentiation within a graph-reduction machine is that the functions to be differentiated are already represented structurally. Rall (1981) emphasizes the importance of the "coder" module in the CODEX system. In reduction systems, the applicative structure of expressions is directly accessible.

Methods like Shearer and Wolfe's ALGLIB (1985) system, intended to operate within a high-level language, also require that transformations be made between string and graph representations of the functions.

Summary

It is feasible to include a differentiation primitive in λ-calculus reduction languages. It effects a symbolic manipulation within a computation that also includes numeric evaluations.

Expressions containing bound variables, can be handled reasonably. λ-abstractions enable the differentiation primitive to act as a true functional. This representation facilitates chain differentiation.

We define the derivative of an abstraction to be the abstraction that represents the function which is the derivative of the input function wrt its bound variable. For an n-ary function it is the list of partial derivatives wrt each of its parameters.

An interesting aspect of our method is that it applies the chain rule to an expression before reducing β-redexes. This provides yet another example of the feasibility of delayed evaluation. Indeed, the chain rule itself is a naturally lazy method of evaluation.

A great opportunity exists for performing symbolic manipulation within a graph-reduction machine, since expressions are already represented in a tree structure. This permits operators that change the structure of a program during its execution.

REFERENCES

Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Comm. ACM, Vol. 21, No. 2, 1978.

Barton, D. and Fitch, J. P. "A Review of Algebraic Manipulation Programs and Their Application," Computer Journal, Vol. 15, 1972.

Brown, W. S., and Hearn, A. C. "Application of Symbolic Mathematical Computations," Computer Physics Communications, Vol. 17, 1979.

Flaherty T. An Implementation of Differentiation as a Built-In Function for a Lambda-Calculus Graph-Reduction Machine. Ph.D. Thesis. Tulane University. New Orleans, Louisiana. 1986.

Golden, J.P. "Differentiation of Unknown Functions in Macsyma," SIGSAM Bulletin. May 1985.

van Hulzen, J. A. and Calmet, J. "Computer Algebra Systems," in Computer Algebra: Symbolic and Algebraic Computation, ed. Buchberger, B., Collins, G. E. and Loos, R. Springer, Wien, 1983.

Landin, P. J. "The Mechanical Evaluation of Expressions," Computer Journal, Vol. 6, 1964.

McCarthy, J. "Recursive functions of symbolic expressions and their computation by machine," Comm. ACM, Vol. 3, No. 4, 1960.

Perrot, J-F. "LISP et Lambda-Calcul," in Lambda-Calcul et Semantique Formelle des Langages de Programmation, Paris, mai, 1979.

Rall, L. B. Automatic Differentiation: Techniques and Applications, Springer-Verlag, Berlin, 1981.

Revesz, G. "An Extension of Lambda-Calculus for Functional Programming," Journal of Logic Programming, Vol. 1, No. 3, 1984.

Revesz, G. "Axioms for the theory of lambda-conversion," SIAM J. Computing, Vol. 14, No. 2, May 1985.

Shearer, J.M. and Wolfe, M. A. "ALGLIB, A Simple Symbol-Manipulation Package," Comm. of ACM, Vol. 28, No. 8, 1985.

Turner, D. A. "A New Implementation Technique for Applicative Languages," Software Practice and Experience, Vol. 9, 1979.

Wester, M. and Steinberg, S. "An Extension to MACSYMA's Concept of Functional Differentiation," SIGSAM Bulletin. August and November, 1983.