# Computational Divided Differencing and Divided-Difference Arithmetics

THOMAS W. REPS and LOUIS B. RALL
University of Wisconsin

Tools for *computational differentiation* transform a program that computes a numerical function $F(x)$ into a related program that computes $F'(x)$ (the derivative of $F$). This paper describes how techniques similar to those used in computational-differentiation tools can be used to implement other program transformations—in particular, a variety of transformations for *computational divided differencing*. We discuss how computational divided-differencing techniques could lead to faster and more robust programs in scientific and graphics applications. We also describe how these ideas relate to the numerical-finite-differencing techniques that motivated Robert Paige's work on finite differencing of set-valued expressions in SETL programs.

*Dedicated to the memory of Robert Paige, 1947-1999.*

## 1. INTRODUCTION

A variety of studies in the field of programming languages have led to useful, high-level transformations that manipulate programs in semantically meaningful ways. In very general terms, these tools transform a program that performs a computation $F(x)$ into a program that performs a related computation $F^{\#}(x)$, for a variety of $F^{\#}$'s of interest.[1] (In some cases, an appropriate preprocessing operation $h$ needs to be applied to the input; in such cases, the transformed program $F^{\#}$ is used to perform a computation of the form $F^{\#}(h(x))$.) Examples of such tools include *partial evaluators* and *program slicers*:

- A *partial evaluator* creates a specialized version of a program when only part of the program's input has been supplied [21,10,34]. A partial evaluator transforms a program that performs a computation $F(\langle s, d \rangle)$, where $F$ operates on a pair of inputs $\langle s, d \rangle$; when $s$ is known, partial evaluation of the program with respect to $s$ results in a program that computes the function $F^s(d)$ $(= F^s(second(\langle s, d \rangle)))$, such that

This work was supported in part by the National Science Foundation under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, and by a Vilas Associate Award from the University of Wisconsin.

Authors' addresses: Thomas W. Reps, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706; Louis B. Rall, Department of Mathematics, University of Wisconsin, 480 Lincoln Dr., Madison, WI 53706.
E-mail addresses: reps@cs.wisc.edu; rall@math.wisc.edu.

[1]In this paper, we do not generally make a distinction between *programs* and *procedures*. We use "program" both to refer to the program as a whole, as well as to refer to individual subroutines in a generic sense. We use "procedure" only in places where we wish to emphasize that the focus of interest is an individual subroutine *per se*.

$$F^s(d) = F(\langle s, d \rangle). \tag{1.1}$$

(The mnemonic is that $s$ and $d$ stand for the “static” and “dynamic” parts of the input, respectively.)

Partial evaluation is useful for removing interpretive overhead, and can also speed up programs that have two arguments that change value at different rates (such as ray tracing [48]).

• The *slice* of a program with respect to a set of program elements $S$ is a projection of the program that includes only program elements that might affect (either directly or transitively) the values of the variables used at members of $S$ [67,51,31]. Given a program that computes a function $F$, one version of the slicing problem focuses on creating the slice by symbolically composing the original program with an appropriate *projection function* $\pi$, where $\pi$ characterizes what part of $F$’s output should be discarded and what part should be retained [63]. Program slicing creates a program that computes $F^\pi$, where

$$F^\pi(x) = (\pi \circ F)(x). \tag{1.2}$$

*Program-slicing* tools allow one to find semantically meaningful decompositions of programs, where the decompositions consist of elements that are not textually contiguous. Slicing, and subsequent manipulation of slices, has applications in many software-engineering tools, including ones for program understanding, maintenance [22], debugging [45], testing [6,3], differencing [30,32], specialization [63], reuse [49], and merging [30].

Less well known in the programming-languages community is the work that has been done by numerical analysts on tools for *computational differentiation* (also known as *automatic differentiation* or *algorithmic differentiation*) [68,56,26,5,27]:

• Given a program that computes a numerical function $F(x)$, a computational-differentiation tool creates a related program that computes $F'(x)$ (the derivative of $F$).

Applications of computational differentiation include optimization, solving differential equations, curve fitting, sensitivity analysis, and many others.

Although in each of the cases mentioned above, the appropriate restructuring of the program could be carried out by hand, hand transformation of a program is an error-prone process. The automated assistance that the aforementioned tools provide prevents errors from being introduced when these transformations are applied.

The work described in this paper expands the set of tools that programmers have at their disposal for performing such high-level, semantically meaningful program manipulations. In particular, we describe how techniques similar to those that have been developed for computational differentiation can be used to transform programs that compute numerical functions into ones that compute divided differences. We also discuss how computational divided-differencing techniques can be used to create more robust algorithms for manipulating representations of curves and surfaces.

The specific contributions of the paper are as follows:

• We present a program transformation that, given a numerical function $F(x)$ defined by a program, creates a program that computes $F[x_0, x_1]$, the *first divided difference* of $F(x)$, where

$$F[x_0, x_1] =_{df} \frac{F(x_0) - F(x_1)}{x_0 - x_1}.$$

Furthermore, we show how computational first divided differencing *generalizes* computational differentiation.

- We present a second program transformation that permits the creation of *higher-order divided differences* of a numerical function defined by a program.
- We present a third program transformation that permits higher-order divided differences to be computed more efficiently. This transformation does not apply to all programs; however, we show that there is at least one important situation where this optimization is of use.
- We show how to extend these techniques to handle functions of several variables.
- Finally, we describe how our work on computational differencing relates to the numerical-finite-differencing techniques that motivated Robert Paige's work on finite differencing of set-valued expressions in SETL programs [52,53].

These ideas are illustrated by means of numerous examples, and implementations of them in the form of C++ class definitions ("divided-difference arithmetics") are sketched.

The divided-differencing problems that are addressed in the paper can be viewed as generalizations of problems such as differentiation, computation of Taylor coefficients, *etc*. Consequently, some of the techniques introduced in this paper—in particular, the organization of the divided-difference arithmetic presented in Section 7—represent new approaches that, with appropriate simplification, can also be applied in computational-differentiation tools.

The remainder of the paper is organized into eight sections: To make the paper self-contained, Section 2 provides a succinct review of the basic principles of computational differentiation that are relevant to our work, and of so-called "differentiation arithmetics" [59,60,61]. Section 3 discusses the basic principle behind computational divided differencing. Section 4 shows how computational divided differencing generalizes computational differentiation. Section 5 extends the ideas introduced in Section 3 to higher-order computational divided differencing. Section 6 discusses techniques that apply to a useful special case. Section 7 extends the ideas from Sections 3, 5, and 6 to functions of several variables. Section 8 describes how these ideas relate to the numerical-finite-differencing techniques that motivated Robert Paige's work on finite differencing of set-valued expressions in SETL programs. Section 9 discusses other related work.

## 2. BACKGROUND ON COMPUTATIONAL DIFFERENTIATION

In scientific and graphics applications, computations model continuous phenomena. Unfortunately, operations in computers are discrete: Instead of providing *real* numbers, computers offer only *floating-point* numbers. This leads to two kinds of errors: (i) *discretization* error, and (ii) *round-off* error. Discretization error is well understood by numerical analysts, and there are well-known techniques for addressing it. However, round-off error is much less well understood, as indicated by the following two statements, made twenty years apart:

> In general, the subject of round-off-error propagation is poorly understood, and very few theoretical results are available [13, pp. 362].

A convenient fiction is that a computer's floating-point arithmetic is "accurate enough." . . . [This] is naive. Notwithstanding, it is a fiction necessarily adopted throughout most of this book. To do a good job of answering the question of how roundoff error propagates, or can be bounded . . . would not be possible: Rigorous analysis of many practical algorithms has never been made, by us or anyone [55, pp. 889-890].

The crux of the problem is the possibility of *catastrophic cancellation*—*i.e.*, the result of an operation may have no significant digits at all.

This problem arises, for instance, in computing values of a function's derivative. Suppose that you have a program `F(x)` that computes a numerical function $F(x)$.[2] It is a very bad idea to try to compute $F'(x_0)$, the value of the derivative of $F$ at $x_0$, by picking a small value `delta_x` and invoking the following program with the argument `x0`:[3]

```
float delta_x = . . . ⟨some small value⟩ . . . ;

float F′_naive(float x){
  return (F(x + delta_x) − F(x))/delta_x;
}
```
(2.1)

For a small enough value of `delta_x`, the values of `F(x0+delta_x)` and `F(x0)` will usually be very close; the catastrophic cancellation that ensues is magnified by dividing by the small quantity `delta_x`, yielding a useless result.

One bright spot during the last thirty years with respect to the control of round-off error has been the emergence of tools for *computational differentiation*, which transform a program that computes a numerical function $F(x)$ into a related program that computes the derivative $F'(x)$.[4] We can illustrate computational differentiation by means of the following example (due to R. Zippel [69]):

EXAMPLE 2.2. Suppose that we have been given a collection of programs `fi` for the functions $f_i$, $1 \le i \le k$, together with the program `Prod` shown below, which computes the function $Prod(x) = \prod_{i=1}^{k} f_i(x)$. In addition, suppose that we have also been given programs `fi′` for the functions $f_i'$, $1 \le i \le k$. Finally, suppose that we wish to obtain a program `Prod′` that computes the function $Prod'(x)$. Row one of the table given below shows mathematical expressions for $Prod(x)$ and $Prod'(x)$. Row two shows two C++ procedures: Procedure `Prod` computes $Prod(x)$; procedure `Prod′` is the procedure that a computational-differentiation system would create to compute $Prod'(x)$.

---

[2]Throughout the paper, `Courier Font` is used to denote functions defined by programs, whereas *Italic Font* is used to denote mathematical functions. That is, $F(x)$ denotes a function (evaluated over real numbers), whereas `F(x)` denotes a program (evaluated over floating-point numbers). We adhere to this convention both in concrete examples that involve C++ code, as well as in more abstract discussions in order to distinguish between a mathematical function and a program that implements the function.

[3]The example programs in the paper are all written in C++; however, the ideas described apply to other programming languages—including functional programming languages, as well as other imperative languages. (To emphasize the links between mathematical concepts and their implementations in C++, we take the liberty of sometimes using ′ and/or subscripts on C++ identifiers.)

[4]Another bright spot has been the application of interval arithmetic to the verification of the accuracy of computed results, for many basic numerical computations [28,29].

| | Function | Derivative |
|---|---|---|
| Mathematical notation | $Prod(x) = \prod_{i=1}^{k} f_i(x)$ | $Prod'(x) = \sum_{i=1}^{k} f_i'(x) * \prod_{j \neq i} f_j(x)$ |
| Programming notation | `float Prod(float x){`<br>`  float ans = 1.0;`<br>`  for (int i = 1; i <= k; i++){`<br>`    ans = ans * f_i(x);`<br>`  }`<br>`  return ans;`<br>`}` | `float Prod′(float x){`<br>`  float ans′ = 0.0;`<br>`  float ans = 1.0;`<br>`  for (int i = 1; i <= k; i++){`<br>`    ans′ = ans′ * f_i(x) + ans * f_i′(x);`<br>`    ans = ans * f_i(x);`<br>`  }`<br>`  return ans′;`<br>`}` |

Notice that program `Prod′` resembles program `Prod`, as opposed to `F′_naive` (see box (2.1)). □

The transformation illustrated above is merely one instance of a general transformation that can be applied to any program: Given a program `G` as input, the transformation produces a derivative-computing program `G′`. The method for constructing `G′` is as follows:

(i) For each variable `v` of type `float` used in `G`, another `float` variable `v′` is introduced.

(ii) Each statement in `G` of the form "`v` = *exp*`;`", where exp is an arithmetic expression, is transformed into "`v′` = *exp′*`;` `v` = *exp*`;`", where *exp′* is the expression for the derivative of *exp*. If *exp* involves calls to a function `g`, then *exp′* may involve calls to both `g` and `g′`.

(iii) Each return statement in `G` of the form "`return v;`" is transformed into "`return v′;`".

In general, this transformation can be justified by appealing to the chain rule of differential calculus.

EXAMPLE 2.3. For Example 2.2, we can demonstrate the correctness of the transformation by symbolically executing `Prod′` for a few iterations, comparing the values of `ans′` and `ans` (as functions of `x`) at the start of each iteration of the for-loop:

| Iteration | Value of `ans′` (as a function of `x`) | Value of `ans` (as a function of `x`) |
|---|---|---|
| 0 | 0.0 | 1.0 |
| 1 | $f_1'(x)$ | $f_1(x)$ |
| 2 | $f_1'(x) * f_2(x) + f_1(x) * f_2'(x)$ | $f_1(x) * f_2(x)$ |
| 3 | $f_1'(x) * f_2(x) * f_3(x) + f_1(x) * f_2'(x) * f_3(x) + f_1(x) * f_2(x) * f_3'(x)$ | $f_1(x) * f_2(x) * f_3(x)$ |
| . . . | . . . | . . . |
| $k$ | $\sum_{i=1}^{k} f_i'(x) * \prod_{j \neq i} f_j(x)$ | $\prod_{i=1}^{k} f_i(x)$ |

The loop maintains the invariant that, at the start of each iteration, $ans'(x) = \frac{d}{dx} ans(x)$.[5] □

[5]The value of `ans′` on the 3rd iteration would actually be computed with the terms grouped as follows: $(f_1'(x) * f_2(x) + f_1(x) * f_2'(x)) * f_3(x) + (f_1(x) * f_2(x)) * f_3'(x)$. Terms have been expanded in the table given above to clarify how `ans′(x)` builds up a value equivalent to $Prod'(x) = \sum_{i=1}^{k} f_i'(x) * \prod_{j \neq i} f_j(x)$.

For the computational-differentiation approach, we did not really need to make the assumption that we were given programs $\mathtt{f}_{\mathtt{i}}'$ for the functions $f_i'$, $1 \le i \le k$; instead, the programs $\mathtt{f}_{\mathtt{i}}'$ can be generated from the programs $\mathtt{f}_{\mathtt{i}}$ by applying the same statement-doubling transformation that was applied to `Prod`.

In languages that support operator overloading, such as C++, Ada, and Pascal-XSC, computational differentiation can be carried out by defining a new data type that has fields for both the value and the derivative, and overloading the arithmetic operators to carry out appropriate manipulations of both fields [57,58], along the lines of the C++ class definition shown below:

```
enum ArgDesc { CONST, VAR };

class FloatD {
 public:
  float val′;
  float val;
  FloatD(ArgDesc,float);
};

// Constructor to convert a constant
// or a value for the independent
// variable to a FloatD
FloatD::FloatD(ArgDesc a, float v){
  switch (a) {
    case CONST:
      val′ = 0.0;
      val = v;
    break;
    case VAR:
      val′ = 1.0;
      val = v;
    break;
  }
}
```

```
FloatD operator+(FloatD a, FloatD b){
  FloatD ans;
  ans.val′ = a.val′ + b.val′;
  ans.val = a.val + b.val;
  return ans;
}

FloatD operator*(FloatD a, FloatD b){
  FloatD ans;
  ans.val′ = a.val * b.val′ + a.val′ * b.val;
  ans.val = a.val * b.val;
  return ans;
}
```

A class such as `FloatD` is called a *differentiation arithmetic* [59,60,61].

The transformation then amounts to changing the types of each procedure's formal parameters, local variables, and return value (including those of the $\mathtt{f}_{\mathtt{i}}$).

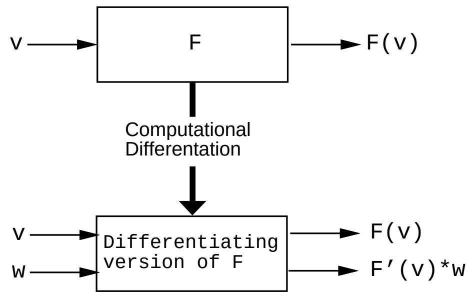
EXAMPLE 2.4. Using class `FloatD`, the `Prod` program of Example 2.2 can be handled as follows:

```
float f1(float x){...}            ⇒   FloatD f1(const FloatD &x){...}
       .                                     .
       .                                     .

float fk(float x){...}            ⇒   FloatD fk(const FloatD &x){...}

float Prod(float x){                  FloatD Prod(const FloatD &x){
  float ans = 1.0;                      FloatD ans(CONST,1.0);  // ans = 1.0
  for (int i = 1; i <= k; i++){         for (int i = 1; i <= k; i++){
    ans = ans * fi(x);            ⇒       ans = ans * fi(x);
  }                                     }
  return ans;                           return ans;
}                                     }

                                      float Prod′(float x){
                                  ⇒     FloatD xD(VAR,x);
                                        return Prod(xD).val′;
                                      }
```

By changing the types of the formal parameters, local variables, and the return values of `Prod` and the $\mathtt{f_i}$ (and making a slight change to the initialization of `ans` in `Prod`), the program now carries around derivative values (in the `val′` field) in addition to performing all of the work performed by the original program. Because of the C++ overload-resolution mechanism, the $\mathtt{f_i}$ procedures invoked in the fourth line of the transformed version of `Prod` are the *transformed* versions of the $\mathtt{f_i}$ (*i.e.*, the $\mathtt{f_i}$ of type `FloatD` → `FloatD`).

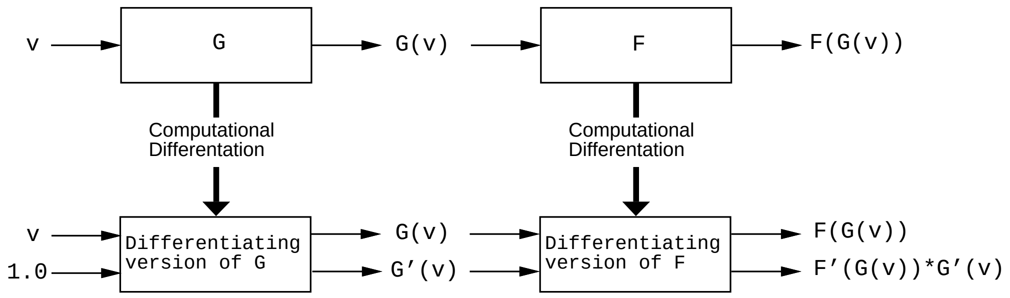
The value of `Prod`'s derivative at `v` is obtained by calling `Prod′(v)`. □

In a differentiation arithmetic, each procedure in the user's program, such as `Prod` and the $\mathtt{f_i}$ in Example 2.4, can be viewed as a box that maps two inputs to two outputs, as depicted below:



In particular, in each differentiating version of a user-defined or library procedure `F`, the lower-right-hand output produces the value `F′(v)*w`.

An input value `v` for the formal parameter is treated as a pair `(v,1.0)`. Boxes like the one shown above "snap together": when `F` is composed with `G` (and the input is `v`), the output value on the lower-right-hand side is `F′(G(v))*G′(v)`, which agrees with the usual expression for the chain rule for the first-derivative operator:


v
G
G(v)
F
F(G(v))
Computational Differentation
Computational Differentation
v
1.0
Differentiating version of G
G(v)
G′(v)
Differentiating version of F
F(G(v))
F′(G(v))*G′(v)

The computational-differentiation technique summarized above is what is known as *forward-mode* differentiation. A different computational-differentiation technique, *reverse mode* [37,65,33,24,25], is generally preferable when the number of independent variables is much greater than the number of dependent variables. However, although it is possible to develop a reverse-mode version of computational divided differencing, it does not appear to offer the same potential savings in operations performed that reverse mode achieves for computational differentiation. Because the forward-mode versions of these operations are somewhat more intuitive than their reverse-mode counterparts, the remainder of the paper concerns the generalization of forward-mode computational differentiation to forward-mode computational divided differencing, and hence reverse mode is not summarized here.

The availability of overloading makes it possible to implement (forward-mode) computational differentiation conveniently, by packaging it as a differentiation-arithmetic class, as illustrated above. The alternative to the use of overloading is to build a special-purpose preprocessor to carry out the statement-doubling transformation that was illustrated in Examples 2.2 and 2.3. Examples of systems that use the latter approach include ADIFOR [7,8] and ADIC [9].[6]

## Limitations of Computational Differentiation

The reader should understand that computational differentiation has certain limitations. One limitation comes from the fact that a program `F′(x)` that results from computational differentiation can perform additions and subtractions for which there are no analogues in the original program `F(x)`. For instance, in program `Prod′`, an addition is performed in the statement

```
ans′ = ans′ * f_i(x) + ans * f_i′(x);
```

whereas no addition is performed in the statement

```
ans = ans * f_i(x);
```

Consequently, the result of evaluating `F′(x)` can be degraded by round-off error even when `F(x)` is computed accurately. However, the accuracy of the result from evaluating `F′(x)` can be verified by performing

[6]We have referred to both computational differentiation and computational divided differencing as “program transformations”, which may conjure up the image of tools that perform source-to-source rewriting fully automatically. Although such tools are one possible embodiment, the reader should understand that in this paper we use the term “transformation” more loosely. For us, “transformation” also includes simpler approaches, including the use of C++ differentiation-arithmetic and divided-difference-arithmetic classes. With these approaches, rewriting might be carried out by a preprocessor, but might also be performed by hand, since usually only light rewriting of the program source text is required.

the same computation in interval arithmetic [57,61].

Another problem that arises is that the manner in which a function is programmed influences whether the results obtained from the derivative program are correct. For instance, for programs that use a conditional expression or conditional statement in which the condition depends on the independent variable—*i.e.*, where the function is defined in a piecewise manner—the derivative program may not produce the correct answer.

EXAMPLE 2.5 [17]. Suppose that the function $F(x) = x^2$ is programmed using a conditional statement, as shown below on the left:

```
float F(float x){
  float ans;
  if (x == 1.0){
    ans = 1.0;
  }
  else{
    ans = x*x;
  }
  return ans;
}
```

$\Longrightarrow$

```
float F′(float x){
  float ans′;
  float ans;
  if (x == 1.0){
    ans′ = 0.0;
    ans = 1.0;
  }
  else{
    ans′ = x+x;
    ans = x*x;
  }
  return ans′;
}
```

Computational differentiation would produce the program shown above on the right. With this program, `F′(1.0)` returns `0.0`, rather than the correct value of `2.0` (*i.e.*, correct with respect to the meaning of the program as the mathematical function $F(x) = x^2$). □

The phenomenon illustrated in Example 2.5 has been called the *branch problem* or the *if problem* for computational differentiation. A more important example of the branch problem occurs in Gaussian elimination code, where pivoting introduces branches into the program [17,4,27]. Some additional problems that can arise with computational differentiation are identified in [17]. A number of different approaches to these problems have been discussed in the literature [17,4,64,36,27].

Computational divided differencing has some similar (or even worse) problems. All of these issues are outside the scope of the present paper; the problem of finding appropriate ways to generalize the aforementioned techniques to handle the problems that arise with computational divided differencing is left for future work.

## 3. COMPUTATIONAL DIVIDED DIFFERENCING

In this paper, we exploit the principle on which computational differentiation is based—namely, that it is possible to differentiate entire *programs*, not just *expressions*—to develop a variety of new *computational divided-differencing* transformations. We develop several transformations that can be applied to numerical programs. One of these corresponds to the *first-divided-difference operator*, denoted by $\cdot[x_0, x_1]$ and defined as follows:

$$F[x_0, x_1] \;=_{df}\; \frac{F(x_0) - F(x_1)}{x_0 - x_1}\,. \tag{3.1}$$

As with the differentiation operator, the problem that we face is that because division by a small value and subtraction are both operations that amplify accumulated round-off errors, direct use of Eqn. (3.1) may lead to highly inaccurate results.

In contrast, given a program that computes a numerical function $F(x)$, our technique for computational first divided differencing creates a related program that computes $F[x_0, x_1]$, *but without directly evaluating the right-hand side of Eqn. (3.1)*. As we show below, the program transformation that achieves this goal is quite similar to the transformation used in computational-differentiation tools. The transformation—defined below—sidesteps the explicit subtraction and division operations that appear in Eqn. (3.1), while producing answers that are equivalent (from the standpoint of evaluation in real arithmetic). The program that results thereby avoids many potentially catastrophic cancellations, and hence retains accuracy when evaluated in floating-point arithmetic.

There are a number of reasons to be interested in this approach:

- Divided differences are the basis for a wide variety of numerical techniques, including algorithms for polynomial interpolation, numerical integration, and solving differential equations [13].
- Finite differences on an evenly spaced grid (which can be obtained from divided differences on an evenly spaced grid) can be used to quickly generate a function's values at any number of points that extend the grid. This technique is useful, therefore, for quickly and accurately plotting a function (see [23] and [53, pp. 403−404]).

In general, the program transformations that we have developed permit the creation of *faster* and *more robust* scientific programs and graphics utilities. (Empirical results presented in Sections 5 and 8 provide two concrete demonstrations of the advantages of these methods.)

To understand the basis of the idea, consider the case in which $F(x) = x^2$:

$$F[x_0, x_1] = \frac{F(x_0) - F(x_1)}{x_0 - x_1} = \frac{x_0^2 - x_1^2}{x_0 - x_1} = x_0 + x_1. \tag{3.2}$$

That is, the first divided difference can be obtained by evaluating $x_0 + x_1$.

Turning to programs, suppose that we are given the following program for squaring a number:

```
float Square(float x){
  return x * x;
}
```

The above discussion implies that to compute the first divided-difference of `Square`, we have our choice between the programs `Square_1DD_naive` and `Square_1DD`:

```
float Square_1DD_naive(float x0,float x1){
  return (Square(x0)−Square(x1))/(x0−x1);
}
```

```
float Square_1DD(float x0,float x1){
  return x0 + x1;
}
```

However, the round-off-error characteristics of `Square_1DD` are much better than those of `Square_1DD_naive`.

In general, for monomials we have:

| $F(x)$ | $c$ | $x$ | $x^2$ | $x^3$ | ... |
|---|---|---|---|---|---|
| $F[x_0, x_1]$ | 0 | 1 | $x_0 + x_1$ | $x_0^2 + x_0 x_1 + x_1^2$ | ... |

The basis for creating expressions and programs that compute accurate divided differences is to be found in the basic properties of the first-divided-difference operator, which closely resemble those of the first-derivative operator, as shown below in Table 3.3:

| First Derivative | First Divided Difference |
|---|---|
| $c' = 0.0$ | $c[x_0, x_1] = 0.0$ |
| $x' = 1.0$ | $x[x_0, x_1] = 1.0$ |
| $(c + F)'(x) = F'(x)$ | $(c + F)[x_0, x_1] = F[x_0, x_1]$ |
| $(c * F)'(x) = c * F'(x)$ | $(c * F)[x_0, x_1] = c * F[x_0, x_1]$ |
| $(F + G)'(x) = F'(x) + G'(x)$ | $(F + G)[x_0, x_1] = F[x_0, x_1] + G[x_0, x_1]$ |
| $(F * G)'(x) = F'(x) * G(x) + F(x) * G'(x)$ | $(F * G)[x_0, x_1] = F[x_0, x_1] * G(x_1) + F(x_0) * G[x_0, x_1]$ |
| $\left(\frac{F}{G}\right)'(x) = \frac{F'(x)^* \; G(x) - F(x)^* \; G'(x)}{G(x)^2}$ | $\left(\frac{F}{G}\right)[x_0, x_1] = \frac{F[x_0, x_1]^* \; G(x_1) - F(x_0)^* \; G[x_0, x_1]}{G(x_0)^* \; G(x_1)}$ |

Table 3.3. Basic properties of the first-derivative and first-divided-difference operators.

The general transformation for computational divided differencing of programs can be explained by means of an example. (As a running example, the paper uses the evaluation of a polynomial in `x` via Horner's rule; *i.e.*, the evaluation procedure accumulates the answer by repeatedly multiplying by `x` and adding in the current coefficient, iterating down from the high-order coefficient. It is well-known that Horner's rule can return inaccurate results when it is used to evaluate a polynomial in floating-point arithmetic [29, pp. 65−67]. Our examples are not meant to illustrate a way to circumvent this shortcoming. Horner's rule is used as the basis of our examples solely because of one virtue: it is a very simple procedure, which allows the various different computational-divided-differencing transformations to be illustrated in a succinct manner.)

EXAMPLE 3.4. Suppose that we have a C++ class `Poly` that represents polynomials, and a member function `Poly::Eval` that evaluates a polynomial via Horner's rule (*i.e.*, it accumulates the answer by repeatedly multiplying by `x` and adding in the current coefficient, iterating down from the high-order coefficient):

```
class Poly {
 public:
  float Eval(float);
 private:
  int degree;
  // Array coeff[0..degree]
  float *coeff;
};
```

```
// Evaluation via Horner’s rule
float Poly::Eval(float x){
  float ans = 0.0;
  for (int i = degree; i >= 0; i−−){
    ans = ans * x + coeff[i];
  }
  return ans;
}
```

A new member function, `Poly::Eval_1DD`, to compute the first divided difference can be created by transforming `Poly::Eval` as shown below:

```
class Poly {
 public:
  float Eval(float);
  float Eval_1DD(float,float);
 private:
  int degree;
  // Array coeff[0..degree]
  float *coeff;
};
```

```
float Poly::Eval_1DD(float x0,float x1){
  float ans_1DD = 0.0;
  float ans = 0.0;
  for (int i = degree; i >= 0; i−−){
    ans_1DD = ans_1DD * x1 + ans;
    ans = ans * x0 + coeff[i];
  }
  return ans_1DD;
}
```

□

The transformation used to obtain `Eval_1DD` from (the text of) `Eval` is similar to the computational-differentiation transformation used to create the derivative-computing program `Eval′`:

- `Eval_1DD` is supplied with an additional formal parameter (and the two parameters are renamed `x0` and `x1`).
- For each local variable `v` of type `float` used in `Eval`, an additional `float` variable `v_1DD` is introduced in `Eval_1DD`.
- Each statement of the form “`v` = *exp*;” in `Eval` is transformed into “`v_1DD` = *exp*[x0,x1]; `v` = $exp_0$;”, where *exp*[x0,x1] is the expression for the divided difference of *exp*, and $exp_0$ is *exp* with `x0` substituted for all occurrences of `x`.
- Each statement of the form “`return v`” in `Eval` is transformed into “`return v_1DD`”.

One caveat concerning the transformation presented above should be noted: the transformation applies only to procedures that have a certain special syntactic structure—namely, the only multiplication operations must be multiplications on the right by the independent variable `x`. Procedure `Eval` is an example of a procedure that has this property.

It is possible to give a fully general first-divided-difference transformation; however, this transformation can be viewed as a special case of the material presented in Section 5. Consequently, we will not pause to present the general first-divided-difference transformation here.

Alternatively, as with computational differentiation, for languages that support operator overloading, computational divided differencing can be carried out with the aid of a new class, say `Float1DD`, for which the arithmetic operators are appropriately redefined. (We will call such a class a *divided-difference arithmetic*.) Computational divided differencing is then carried out by making appropriate changes to the types of each procedure's formal parameters, local variables, and return value.

Again, definitions of first-divided-difference arithmetic classes—both for the case of general first divided differences, as well as for the special case that covers programs like `Eval`—can be viewed as special cases of the divided-difference arithmetic classes `FloatDD` and `FloatDDR1` discussed in Sections 5 and 6, respectively. For this reason, we postpone giving a concrete example of a divided-difference arithmetic until Sections 5 and 6.

## 4. COMPUTATIONAL DIVIDED DIFFERENCING AS A GENERALIZATION OF COMPUTATIONAL DIFFERENTIATION

In this section, we explain in what sense computational divided differencing can be said to generalize computational differentiation. First, observe that over real numbers, we have

$$\lim_{x_1 \to x_0} F[x_0, x_1] = \lim_{x_1 \to x_0} \frac{F(x_0) - F(x_1)}{x_0 - x_1} = F'(x_0). \tag{4.1}$$

However, although $\lim_{x_1 \to x_0} \frac{F(x_0) - F(x_1)}{x_0 - x_1} = F'(x_0)$ holds over reals, it does not hold over floating-point numbers: as $\mathtt{x}_1$ approaches $\mathtt{x}_0$, because of accumulated round-off errors and catastrophic cancellation, the quantity $\frac{\mathtt{F}(\mathtt{x}_0) - \mathtt{F}(\mathtt{x}_1)}{\mathtt{x}_0 - \mathtt{x}_1}$ *does not*, in general, approach $\mathtt{F}'(\mathtt{x}_0)$.[7] This is why derivatives cannot be computed accurately by procedure `F'_naive` (see box (2.1)).

In contrast, as a floating-point value $\mathtt{x}_1$ approaches $\mathtt{x}_0$, the result computed by the *program* obtained via the computational-divided-differencing transformation approaches the result computed by the *program* obtained via the computational-differentiation transformation. That is, letting the terms $\mathtt{F}[\mathtt{x}_0,\mathtt{x}_1]$ and $\mathtt{F}'(\mathtt{x}_0)$ stand for these programs, respectively, we have

$$\lim_{\mathtt{x}_1 \to \mathtt{x}_0} \mathtt{F}[\mathtt{x}_0,\mathtt{x}_1] = \mathtt{F}'(\mathtt{x}_0). \tag{4.2}$$

EXAMPLE 4.3. To illustrate Eqn. (4.2), consider applying the two transformations to member function `Poly::Eval` of our earlier example:

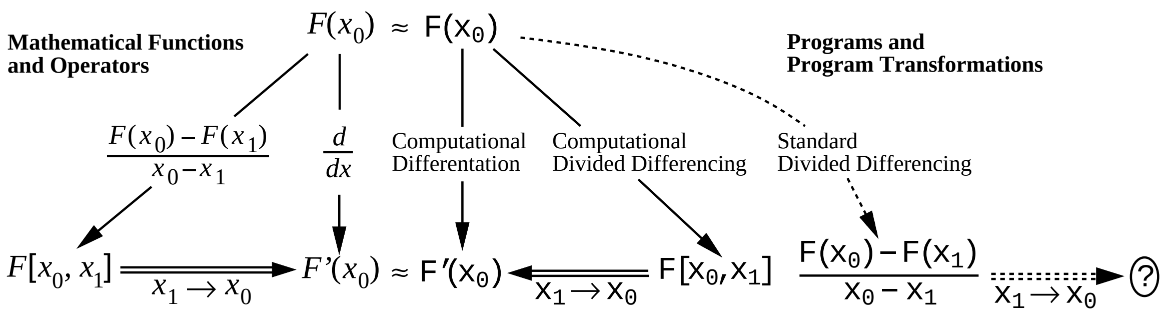
[7]Note the use of `Courier Font` here; we are making a statement about quantities computed by *programs*.

```
class Poly {
 public:
  float Eval(float);
  float Eval_1DD(float,float);
  float Eval′(float);
 private:
  int degree;
  // Array coeff[0..degree]
  float *coeff;
};
```

```
float Poly::Eval_1DD(float x0,float x1){
  float ans_1DD = 0.0;
  float ans = 0.0;
  for (int i = degree; i >= 0; i--){
    ans_1DD = ans_1DD * x1 + ans;
    ans = ans * x0 + coeff[i];
  }
  return ans_1DD;
}
float Poly::Eval′(float x){
  float ans′ = 0.0;
  float ans = 0.0;
  for (int i = degree; i >= 0; i--){
    ans′ = ans′ * x + ans;
    ans = ans * x + coeff[i];
  }
  return ans′;
}
```

When formal parameters `x0`, `x1`, and `x` all have the same value—say `v`—then *exactly the same operations* are performed by `Eval_1DD(v,v)` and `Eval′`(v). □

Because computations are carried out over floating-point numbers, the programs $\mathtt{F[x_0,x_1]}$ and $\mathtt{F'(x_0)}$ are only approximations to the functions that we actually desire. That is, $\mathtt{F[x_0,x_1]}$ approximates the function $F[x_0, x_1]$, and $\mathtt{F'(x_0)}$ approximates $F'(x_0)$. The relationships among these functions and programs are depicted below:


Mathematical Functions and Operators
$F(x_0) \approx \mathtt{F(x_0)}$
Programs and Program Transformations
$\frac{F(x_0)-F(x_1)}{x_0-x_1}$
$\frac{d}{dx}$
Computational Differentation
Computational Divided Differencing
Standard Divided Differencing
$F[x_0, x_1]$
$x_1 \to x_0$
$F'(x_0) \approx \mathtt{F'(x_0)}$
$\mathtt{x_1 \to x_0}$
$\mathtt{F[x_0,x_1]}$
$\frac{\mathtt{F(x_0)-F(x_1)}}{\mathtt{x_0-x_1}}$
$\mathtt{x_1 \to x_0}$
?

As the discussion above has noted, the interesting feature of this diagram is the relationship between $\mathtt{F[x_0,x_1]}$ and $\mathtt{F'(x_0)}$, on the side of the diagram labeled "Programs and Program Transformations". In particular, as $\mathtt{x_1}$ approaches $\mathtt{x_0}$, $\mathtt{F[x_0,x_1]}$ approaches $\mathtt{F'(x_0)}$. Consequently, a program produced by a system for computational divided differencing can be used to compute values of derivatives (in addition to divided differences) by feeding it duplicate actual parameters (*e.g.*, `Eval_1DD(v,v)`). In contrast, a program created by means of computational differentiation can only produce derivatives (and not divided differences). In this sense, computational divided differencing can be said to generalize computational differentiation.

Computational divided differencing suffers from one of the same problems that arises with computational differentiation, namely that the program $\mathtt{F[x_0,x_1]}$ that results from the transformation can perform additions and subtractions that have no analogues in the original program `F(x)`. Consequently, the result of evaluating `F[x`$_0$`,x`$_1$`]` can be degraded by round-off error even when `F(x)` is computed accurately. However, because of the fact that `F[x`$_0$`,x`$_1$`]` converges to `F′(x`$_0$`)` as `x`$_1$ approaches `x`$_0$, if `F′(x`$_0$`)` returns a result of sufficient accuracy, then `F[x`$_0$`,x`$_1$`]` will return a result of sufficient accuracy when $|x_0 - x_1|$ is small. (In future work, we plan to investigate the use of interval arithmetic for providing interval bounds for computational divided differencing.)

## 5. HIGHER-ORDER COMPUTATIONAL DIVIDED DIFFERENCING

In this section, we show that the idea that was introduced in Section 3 can be generalized to define a transformation for *higher-order computational divided differencing*. To do so, we will define a divided-difference arithmetic that manipulates *divided-difference tables*.

Higher-order divided differences are divided differences of divided differences, defined recursively as follows:

$$F[x_i] \quad =_{df} \quad F(x_i) \tag{5.1}$$

$$F[x_0, x_1, \ldots, x_{n-1}, x_n] \quad =_{df} \quad \frac{F[x_0, x_1, \ldots, x_{n-1}] - F[x_1, \ldots, x_{n-1}, x_n]}{x_0 - x_n} \tag{5.2}$$

In our context, a divided-difference table for a function $F$ is an upper-triangular matrix whose entries are divided differences of different orders, as indicated below:[8]

| | | | |
|---|---|---|---|
| $F(x_0)$ | $F[x_0, x_1]$ | $F[x_0, x_1, x_2]$ | $F[x_0, x_1, x_2, x_3]$ |
| 0 | $F(x_1)$ | $F[x_1, x_2]$ | $F[x_1, x_2, x_3]$ |
| 0 | 0 | $F(x_2)$ | $F[x_2, x_3]$ |
| 0 | 0 | 0 | $F(x_3)$ |

Higher-order divided differences have numerous applications in interpolation and approximation of functions [13].

We occasionally use $[x_{i,j}]$ as an abbreviation for $[x_i, \ldots, x_j]$. However, the reader should note that

$$\begin{aligned} F[x_{0,2}] &= F[x_0, x_1, x_2] \\ &= \frac{F[x_0, x_1] - F[x_1, x_2]}{x_0 - x_2}\,, \end{aligned}$$

which is not the same as $F[x_0, x_2]$:

$$F[x_0, x_2] = \frac{F(x_0) - F(x_2)}{x_0 - x_2}\,.$$

[8]Other arrangements of $F$'s higher-order divided differences into matrix form are possible. For example, one could have a lower triangular matrix with the $F(x_i)$ running down the first column. However, the use of the arrangement shown above is key to being able to use simple notation—*i.e.*, ordinary matrix operations—to describe our methods [50].

We use $\cdot^{\triangledown[x_0,\cdots,x_n]}$ to denote the operator that yields the divided-difference table for a function with respect to points $x_0, \cdots, x_n$. (We use $\cdot^{\triangledown}$ if the points $x_0, \cdots, x_n$ are clear from the context.)

A method for creating accurate divided-difference tables for *rational expressions* is found in Opitz [50]. This method is based on the properties of $\cdot^{\triangledown[x_0,\cdots,x_n]}$ given in the right-hand column of Table 5.3:

| First Divided Difference | Divided-Difference Table |
|---|---|
| $c[x_0, x_1] = 0.0$ | $c^{\triangledown[x_0,\cdots,x_n]} = c * I$ |
| $x[x_0, x_1] = 1.0$ | $x^{\triangledown[x_0,\cdots,x_n]} = A_{[x_0,\cdots,x_n]}$ |
| $(c+F)[x_0, x_1] = F[x_0, x_1]$ | $(c+F)^{\triangledown[x_0,\cdots,x_n]} = c * I + F^{\triangledown[x_0,\cdots,x_n]}$ |
| $(c*F)[x_0, x_1] = c * F[x_0, x_1]$ | $(c*F)^{\triangledown[x_0,\cdots,x_n]} = c * F^{\triangledown[x_0,\cdots,x_n]}$ |
| $(F+G)[x_0, x_1] = F[x_0, x_1] + G[x_0, x_1]$ | $(F+G)^{\triangledown[x_0,\cdots,x_n]} = F^{\triangledown[x_0,\cdots,x_n]} + G^{\triangledown[x_0,\cdots,x_n]}$ |
| $(F*G)[x_0, x_1] = F[x_0, x_1] * G(x_1) + F(x_0) * G[x_0, x_1]$ | $(F*G)^{\triangledown[x_0,\cdots,x_n]} = F^{\triangledown[x_0,\cdots,x_n]} * G^{\triangledown[x_0,\cdots,x_n]}$ |
| $\left(\frac{F}{G}\right)[x_0, x_1] = \frac{F[x_0, x_1] * G(x_1) - F(x_0) * G[x_0, x_1]}{G(x_0) * G(x_1)}$ | $\left(\frac{F}{G}\right)^{\triangledown[x_0,\cdots,x_n]} = \frac{F^{\triangledown[x_0,\cdots,x_n]}}{G^{\triangledown[x_0,\cdots,x_n]}}$ |

Table 5.3. Basic properties of two divided-difference operators.

A few items in Table 5.3 require explanation:

(1) The symbol $I$ denotes the identity matrix.

(2) The symbol $A_{[x_0,\cdots,x_n]}$ denotes the matrix

$$\begin{pmatrix} x_0 & 1 & 0 & . & . & . & 0 & 0 \\ 0 & x_1 & 1 & . & . & . & 0 & 0 \\ 0 & 0 & x_2 & . & . & . & 0 & 0 \\ . & . & . & . & & & . & . \\ . & . & . & & . & & . & . \\ . & . & . & & & . & . & . \\ 0 & 0 & 0 & . & . & . & x_{n-1} & 1 \\ 0 & 0 & 0 & . & . & . & 0 & x_n \end{pmatrix}$$

(3) In the entry for $(F*G)^{\triangledown[x_0,\cdots,x_n]}$, the multiplication operation in $F^{\triangledown[x_0,\cdots,x_n]} * G^{\triangledown[x_0,\cdots,x_n]}$ is matrix multiplication.

(4) In the entry for $\left(\frac{F}{G}\right)^{\triangledown[x_0,\cdots,x_n]}$, the division operation in $\frac{F^{\triangledown[x_0,\cdots,x_n]}}{G^{\triangledown[x_0,\cdots,x_n]}}$ is matrix division (*i.e.*, $\frac{P}{Q} = P * Q^{-1}$).

The two columns of Table 5.3 can be read as recursive definitions for the operations $\cdot[x_0, x_1]$ and $\cdot^{\triangledown[x_0,\cdots,x_n]}$, respectively. These have straightforward implementations as recursive programs that walk over an expression tree.

The reader should be aware that the $\cdot[x_0, x_1]$ operation defined in the first column of Table 5.3 creates an expression that computes *only* the first divided difference of the original expression $e(x)$, whereas the operation defined in the second column of Table 5.3 creates a (matrix) expression that computes *all* of the first, second, ..., $n^{th}$ divided differences with respect to the points $x_0$, $x_1$, ..., $x_n$, *as well as* $n+1$ values of $e(x)$.

In particular, in the matrix that results from evaluating the transformed expression, the values of $e(x)$ evaluated at the points $x_0$, $x_1$, ..., $x_n$ are found on the diagonal. For instance, in the case of an expression that multiplies two subexpressions $F(x)$ and $G(x)$, the elements on the diagonal are $F(x_0) * G(x_0)$, $F(x_1) * G(x_1)$, ..., $F(x_n) * G(x_n)$.

It is easy to verify (by means of induction) that the first and second columns of Table 5.3 are consistent with each other: in each case, the quantity $e[x_0, x_1]$ represents the (0,1) entry of the matrix $e^{\triangledown[x_0,\cdots,x_n]}$.

The second column of Table 5.3 has an even more straightforward interpretation:

OBSERVATION 5.4 [Reinterpretation Principle]. The divided-difference table of an arithmetic expression $e(x)$ with respect to the $n+1$ points $x_0, \cdots, x_n$ can be obtained by reinterpreting $e(x)$ as a *matrix expression*, where the matrix $A_{[x_0,\cdots,x_n]}$ is used at each occurrence of the variable $x$, and $c * I$ is used at each occurrence of a constant $c$. □

That is, the expression tree for $e(x)$ is unchanged—except at its leaves, where $A_{[x_0,\cdots,x_n]}$ is used in place of $x$, and $c * I$ is used in place of $c$—but the operators at all internal nodes are reinterpreted as denoting matrix operations. This observation is due to Opitz [50].

With only a slight abuse of notation, we can express this as

$$e^{\triangledown[x_0,\cdots,x_n]} = e(A_{[x_0,\cdots,x_n]}).$$

Using this notation, we can show that the chain rule for the divided-difference operator $\cdot^{\triangledown[x_0,\cdots,x_n]}$ has the following particularly simple form:

$$\begin{aligned} (F \circ G)^{\triangledown[x_0,\cdots,x_n]} &= (F \circ G)(A_{[x_0,\cdots,x_n]}) \\ &= F(G(A_{[x_0,\cdots,x_n]})) \\ &= F(G^{\triangledown[x_0,\cdots,x_n]}). \end{aligned} \tag{5.5}$$

Opitz's idea can be extended to the creation of accurate divided-difference tables for functions defined by *programs* by overloading the arithmetic operators used in the program to be matrix operators—*i.e.*, by defining a divided-difference arithmetic that manipulates divided-difference tables. It is worthwhile giving a name to this principle:

OBSERVATION 5.6 [Computational Divided-Differencing Principle]. Rather than computing a divided-difference table with respect to the points $x_0$, $x_1$, ..., $x_n$ by invoking the program $n+1$ times and then applying Eqns. (5.1) and (5.2), we may instead evaluate the program (once) using a divided-difference arithmetic that overloads arithmetic operations as matrix operations, substituting $A_{[x_0,\cdots,x_n]}$ for each occurrence of the formal parameter $x$, and $c * I$ for each occurrence of a constant $c$. □

The reader should understand that the single invocation of the program using the divided-difference arithmetic will actually be more expensive than the $n+1$ ordinary invocations of the program. The advantage of using divided-difference arithmetic is *not* that execution is speeded up because the program is only invoked once (in fact, execution is slower); the advantage is that the result computed using divided-difference arithmetic is much more *accurate*.

Because higher-order divided differences are defined recursively in terms of divided differences of lower order (*cf*. Eqns. (5.1) and (5.2)), it would be possible to define an algorithm for higher-order computational-divided-differencing using repeated applications of lower-order computational-divided-differencing transformations. However, with each application of the transformation for computational first divided differencing, the program that results performs three times the number of operations that are performed by the program the transformation starts with. Consequently, this approach has a significant drawback: the final program that would be created for computing $k^{th}$ divided differences could be $O(3^k)$ times slower than the original program. In contrast, the slow-down factor with the approach based on Observation 5.6 is $O(k^3)$.

We now sketch how a version of higher-order computational divided differencing based on Observation 5.6 can be implemented in C++. Below, we present highlights of a divided-difference arithmetic class, named `FloatDD`. We actually make use of two classes: (i) class `FloatDD`, the divided-difference arithmetic proper, and (ii) class `FloatV`, vectors of `x`$_i$ values. These classes are defined as follows:

```
class FloatDD {
 public:
  int numPts;
  float **divDiffTable;    // Two-dimensional upper-triangular array

  FloatDD(const FloatV &); // Construct a FloatDD from a FloatV
  FloatDD(int N);          // Construct a zero-valued FloatDD of size N-by-N

  FloatDD& operator+ (const FloatDD &) const; // binary addition
  FloatDD& operator- (const FloatDD &) const; // binary subtraction
  FloatDD& operator* (const FloatDD &) const; // binary multiplication
  FloatDD& operator/ (const FloatDD &) const; // binary division
};
```

```
class FloatV {
 public:
  int numPts;
  float *val;   // An array of values: val[0]..val[numPts−1]

  FloatV(int N, ...);                      // N points
  FloatV(float start, int N, float incr); // N equally spaced points
};
```

The constructor `FloatDD(const FloatV &);` plays the role of generating a matrix $A_{[x_0,\cdots,x_n]}$ from a vector $[x_0, \cdots, x_n]$ of values for the independent variable. It is defined as follows:

```
// Construct a FloatDD from a FloatV
FloatDD::FloatDD(const FloatV &fv) :
  numPts(fv.numPts),
  divDiffTable(calloc_ut(numPts))  // allocate upper-triangular matrix of zeros
{
  for (int i = 0; i < numPts; i++) {
    divDiffTable[i][i] = fv.val[i];
    if (i < numPts−1) {
      divDiffTable[i][i+1] = 1.0;
    }
  }
}
```

The multiplication operator of class `FloatDD` simply performs matrix multiplication:

```
FloatDD& FloatDD::operator* (const FloatDD &fdd) const
{
  assert(numPts == fdd.numPts);

  FloatDD *ans = new FloatDD(numPts);
  for (int r = 0; r < numPts; r++) {
    for (int c = r; c < numPts; c++) {
      float temp = 0.0;
      for (int k = r; k <= c; k++) {
        temp += divDiffTable[r][k] * fdd.divDiffTable[k][c];
      }
      ans−>divDiffTable[r][c] = temp;
    }
  }
  return *ans;
}
```

The division operator of class `FloatDD` is implemented using back substitution. That is, suppose we wish to find the value of `A/B` (call this value `X`). `X` can be found by solving the system `X * B = A`. Because the divided-difference tables `A` and `B` are both upper-triangular matrices, this can be done using back substitution, as follows:

```
// Use back substitution
FloatDD& FloatDD::operator/ (const FloatDD &fdd) const
{
  assert(numPts == fdd.numPts);
  assert(NonZeroDiagonal(fdd));

  FloatDD *ans = new FloatDD(numPts);
  for (int r = 0; r < numPts; r++) {
    for (int c = r; c < numPts; c++) {
      float temp = 0.0;
      for (int k = r; k < c; k++) {
        temp += ans->divDiffTable[r][k] * fdd.divDiffTable[k][c];
      }
      ans->divDiffTable[r][c] =
           (divDiffTable[r][c] - temp) / fdd.divDiffTable[c][c];
    }
  }
  return *ans;
}
```

(5.7)

EXAMPLE 5.8. To illustrate these definitions, consider again the function `Poly::Eval` that evaluates a polynomial via Horner's rule. Computational divided differencing is carried out by changing the types of `Eval`'s formal parameters, local variables, and return value from `float` to `FloatDD`:

```
// Evaluation via Horner's rule
float Poly::Eval(float x){
  float ans = 0.0;
  for (int i = degree; i >= 0; i--){
    ans = ans * x + coeff[i];
  }
  return ans;
}
```

```
// Evaluation via Horner's rule
FloatDD Poly::Eval(const FloatDD &x){
  FloatDD ans(x.numPts);    // ans = 0.0
  for (int i = degree; i >= 0; i--){
    ans = ans * x + coeff[i];
  }
  return ans;
}
```

The transformed procedure can be used to generate the divided-difference table for the polynomial

$$P(x) = 2.1 * x^3 - 1.4 * x^2 - .6 * x + 1.1$$

with respect to the (unevenly spaced) points 3.0, 3.01, 3.02, 3.05 by performing the following operations:

```
Poly *P = new Poly(4,2.1,-1.4,-0.6,1.1);
FloatV x(4,3.0,3.01,3.02,3.05);
FloatDD A(x);  // Corresponds to A[x0,...,x3]
FloatDD fdd = P->Eval(A);
```

Empirical results from calculations of this sort are discussed below. □

## Empirical Results: Higher-Order Divided Differencing

We now present some empirical results that illustrate the advantages of the computational-divided-differencing method. In the experiment reported below, we worked with the polynomial

$$P(x) = 2.1 * x^3 - 1.4 * x^2 - .6 * x + 1.1,$$

and performed computations using single-precision floating-point arithmetic on a Sun SPARCstation 20/61 running SunOS 5.6. Programs were compiled with the egcs-2.91.66 version of `g++` (egcs-1.1.2 release) with optimization at the `-O1` level. The experiment compared the standard method for generating divided-difference tables—namely, the recursive definition given in Eqns. (5.1) and (5.2)—against the overloaded version of function `Poly::Eval` from Example 5.8.

In each of the four examples shown below, the values used for the (unevenly spaced) points $x_0$, $x_1$, $x_2$, and $x_3$ are shown on the left. Note how the standard method for generating divided-difference tables degrades as the points move closer together. (Places where the results from the two methods differ are indicated in boldface.) In particular, because $P$ is a cubic polynomial whose high-order coefficient is 2.1, the proper value of $P[x_0, x_1, x_2, x_3]$—the third divided difference of $P$—is 2.1, not 117,520! (Compare the two entries in the northeast corners of the bottom two divided-difference tables.)

| | Computational Divided Differencing | | | | Standard Divided Differencing | | | |
|---|---|---|---|---|---|---|---|---|
| $x_0$: 3.0 | 43.4 | 67.3 | 23.8 | 2.**1** | 43.4 | 67.3 | 23.8 | 2.**09999** |
| $x_1$: 4.0 | | 110.7 | 114.9 | 32.2 | | 110.7 | 114.9 | 32.2 |
| $x_2$: 5.0 | | | 225.6 | 211.5 | | | 225.6 | 211.5 |
| $x_3$: 7.0 | | | | 648.6 | | | | 648.6 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $x_0$: 3.0 | 43.4 | 47.87**52** | 17.5**63** | **2.1** | 43.4 | 47.87**49** | 17.5**858** | **1.59073** |
| $x_1$: 3.01 | | 43.8787 | 48.226**5** | 17.66**8** | | 43.8787 | 48.226**6** | 17.66**53** |
| $x_2$: 3.02 | | | 44.361 | 48.9332 | | | 44.361 | 48.9332 |
| $x_3$: 3.05 | | | | 45.829 | | | | 45.829 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $x_0$: 3.0 | 43.4 | 47.717**5** | 1**7.5063** | **2.1** | 43.4 | 47.717**7** | **15.2886** | **754.685** |
| $x_1$: 3.001 | | 43.4477 | 47.7**525** | 1**7.5168** | | 43.4477 | 47.7**483** | 1**9.0621** |
| $x_2$: 3.002 | | | 43.4955 | 47.82**26** | | | 43.4955 | 47.82**45** |
| $x_3$: 3.005 | | | | 43.6389 | | | | 43.6389 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $x_0$: 3.0 | 43.4 | 47.**7017** | **17.5006** | **2.1** | 43.4 | 47.**6945** | **3.62336** | **117520** |
| $x_1$: 3.0001 | | 43.4048 | 47.**7052** | **17.5017** | | 43.4048 | 47.**6952** | **62.379** |
| $x_2$: 3.0002 | | | 43.4095 | 47.7**122** | | | 43.4095 | 47.7**202** |
| $x_3$: 3.0005 | | | | 43.4238 | | | | 43.4238 |

Finally, when we set all of the input values to 3.0, we obtain

Computational Divided Differencing

| | | | | |
|---|---|---|---|---|
| $x_0$: 3.0 | 43.4 | **47.7** | **17.5** | **2.1** |
| $x_1$: 3.0 | | 43.4 | **47.7** | **17.5** |
| $x_2$: 3.0 | | | 43.4 | **47.7** |
| $x_3$: 3.0 | | | | 43.4 |

Standard Divided Differencing

| | | | | |
|---|---|---|---|---|
| $x_0$: 3.0 | 43.4 | **NaN** | **NaN** | **NaN** |
| $x_1$: 3.0 | | 43.4 | **NaN** | **NaN** |
| $x_2$: 3.0 | | | 43.4 | **NaN** |
| $x_3$: 3.0 | | | | 43.4 |

With the standard divided-differencing method, division by 0 occurs and yields the exceptional value NaN. In contrast, computational divided differencing produces values for $P$'s first, second, and third derivatives. More precisely, each $k^{th}$ divided-difference entry in the computational-divided-differencing table equals

$$\frac{1}{k!} \left. \frac{d^k \, P(x)}{dx^k} \right|_{x=3.0} \qquad (5.9)$$

The $k = 1$ case was already discussed in Section 4, where we observed that computational first divided differencing could be used to compute first derivatives.

## 6. A SPECIAL CASE

A divided-difference table for a function $F$ can be thought of as a (redundant) representation of an interpolating polynomial for $F$. For instance, if you have a divided-difference table $T$ (and also know the appropriate vector of values $x_0$, $x_1$, ..., $x_n$), you can explicitly construct the Newton form of the interpolating polynomial for $F$ according to the following definition [13, pp. 197]:

$$p_n(x) = \sum_{i=0}^{n} F[x_0, \ldots, x_i] * \prod_{j=0}^{i-1} (x - x_j) \qquad (6.1)$$

Note that to be able to create the Newton form of the interpolating polynomial for $F$ via Eqn. (6.1), only the first row of divided-difference table $T$ is required to be at hand—*i.e.*, the values $F[x_0, \ldots, x_i]$, for $0 \le i \le n$—together with the values of $x_0$, $x_1$, ..., $x_n$. This observation suggests that we should develop an alternative divided-difference arithmetic that builds up and manipulates only first rows of divided-difference tables. We call this divided-difference arithmetic `FloatDDR1` (for *D*ivided-*D*ifference *R*ow *1*). The motivation for this approach is that `FloatDDR1` operations will be much faster than `FloatDD` ones, because `FloatDD` operations must manipulate upper-triangular matrices, whereas `FloatDDR1` operations involve only simple vectors.

To achieve this, we define class `FloatDDR1` as follows:

```
class FloatDDR1 {
  friend FloatDDR1& operator+ (const FloatDDR1 &, const float);
  friend FloatDDR1& operator+ (const FloatDDR1 &, const FloatV &);
  friend FloatDDR1& operator+ (const float,       const FloatDDR1 &);
  friend FloatDDR1& operator+ (const FloatV &,    const FloatDDR1 &);

  friend FloatDDR1& operator- (const FloatDDR1 &, const float);
  friend FloatDDR1& operator- (const FloatDDR1 &, const FloatV &);
  friend FloatDDR1& operator- (const float,       const FloatDDR1 &);
  friend FloatDDR1& operator- (const FloatV &,    const FloatDDR1 &);

  friend FloatDDR1& operator* (const FloatDDR1 &, const float);
  friend FloatDDR1& operator* (const FloatDDR1 &, const FloatV &);
  friend FloatDDR1& operator* (const float,       const FloatDDR1 &);
  friend FloatDDR1& operator* (const FloatV &,    const FloatDDR1 &);

  friend FloatDDR1& operator/ (const FloatDDR1 &, const float);
  friend FloatDDR1& operator/ (const FloatDDR1 &, const FloatV &);
 public:
  int numPts;
  float *divDiffTable; // One-dimensional array of divided differences

  FloatDDR1(int N)     // Construct a zero-valued FloatDDR1 of length N
    : numPts(N), divDiffTable(new float[numPts])
    { }
  FloatDDR1& operator+ (const FloatDDR1 &) const; // binary addition
};
```

Compared with class `FloatDD`, class `FloatDDR1` is somewhat impoverished: we can add two arbitrary `FloatDDR1`'s; however, because we do not have full divided-difference tables available, we cannot multiply two arbitrary `FloatDDR1`'s; nor do we have the full $A_{[x_0,\cdots,x_n]}$ matrices that are used at each occurrence of the independent variable. We finesse these difficulties by limiting the operations of class `FloatDDR1` to the ones indicated in the class definition given above: (i) addition, subtraction, and multiplication on either side by a `float` or a `FloatV`; (ii) division on the right by a `float` or a `FloatV`.

The operations that involve a `float` argument `c` have their "obvious" meanings, if one bears in mind that a `float` value `c` serves as a stand-in for a full matrix `c*I`. For the addition (subtraction) operations, `c` is only added to (subtracted from) the `divDiffTable[0]` entry of the `FloatDDR1` argument. For the multiplication (division) operations, all of the `divDiffTable` entries are multiplied by (divided by) `c`.

In the operations that involve a `FloatV` argument, the `FloatV` value serves as a stand-in for a full $A_{[x_0,\cdots,x_n]}$ matrix. For instance, the operator for multiplication on the right by a `FloatV` can be thought of as performing a form of matrix multiplication—but specialized to produce only the first row of the output divided-difference table (and to use only values that are available in the given `FloatDDR1` and `FloatV` arguments):

```
FloatDDR1& operator* (const FloatDDR1 &fddr1, const FloatV &fv)
{
  FloatDDR1 *ans = new FloatDDR1(fddr1.numPts);
  ans->divDiffTable[0] = fddr1.divDiffTable[0] * fv.val[0];
  for (int c = 1; c < fddr1.numPts; c++) {
    ans->divDiffTable[c] =
          fddr1.divDiffTable[c-1] + fddr1.divDiffTable[c] * fv.val[c];
  }
  return *ans;
}
```

It might be thought that the operator for multiplication on the left by a `FloatV` does not have the proper values available in the given `FloatV` and `FloatDDR1` arguments to produce the first row of the product divided-difference table as output. (In particular, the second argument, which is of type `FloatDDR1`, is a row vector, yet we want to produce a row vector as the result.) However, it is easy to show that divided-difference matrices are commutative:

$$F^{\text{◹}} * G^{\text{◹}} = (F * G)^{\text{◹}} = (G * F)^{\text{◹}} = G^{\text{◹}} * F^{\text{◹}}. \qquad (6.2)$$

Consequently, the operator for multiplication on the left by a `FloatV` can be treated as if the `FloatV` were on the right:

```
FloatDDR1& operator* (const FloatV &fv, const FloatDDR1 &fddr1)
{
  return fddr1 * fv;
}
```

As with class `FloatDD`, the division operator is implemented using a form of back substitution—specialized here to compute just what is needed for the first row of the divided-difference table:

```
FloatDDR1& operator/ (const FloatDDR1 &fddr1, const FloatV &fv)
{
  FloatDDR1 *ans = new FloatDDR1(fddr1.numPts);
  ans->divDiffTable[0] = fddr1.divDiffTable[0] / fv.val[0];
  for (int c = 1; c < fddr1.numPts; c++) {
    ans->divDiffTable[c] =
       (fddr1.divDiffTable[c] - ans->divDiffTable[c-1]) / fv.val[c];
  }
  return *ans;
}
```

Because only a limited set of arithmetic operations are available for objects of class `FloatDDR1`, this divided-difference arithmetic can only be applied to procedures that have a certain special syntactic structure, namely ones that are "accumulative" in the independent variable (with only "right-accumulative" quotients). In other words, the procedure must never perform arithmetic on two local variables that both depend on the independent variable.

EXAMPLE 6.3. The procedure `Poly::Eval` for evaluating a polynomial via Horner's rule is an example of a procedure of the right form. Consequently, an overloaded version of the function `Poly::Eval` using `FloatDDR1` arithmetic can be written as shown below on the right:

| | |
|---|---|
| `// Evaluation via Horner’s rule`<br>`float Poly::Eval(float x){`<br>`  float ans = 0.0;`<br>`  for (int i = degree; i >= 0; i−−){`<br>`    ans = ans * x + coeff[i];`<br>`  }`<br>`  return ans;`<br>`}` | `// Evaluation via Horner’s rule`<br>`FloatDDR1 Poly::Eval(const FloatV &x){`<br>`  FloatDDR1 ans(x.numPts);    // ans = 0.0`<br>`  for (int i = degree; i >= 0; i−−){`<br>`    ans = ans * x + coeff[i];`<br>`  }`<br>`  return ans;`<br>`}` |

In Section 3, Example 3.4 discussed the procedure `Poly::Eval_1DD`, a transformed version of `Poly::Eval` that computes the value of the first divided difference of a polynomial with respect to two values, $x_0$ and $x_1$. With the way that the overloaded operations are defined for class `FloatDDR1`, when the actual parameter supplied for `x` is a `FloatV` of length two consisting of $x_0$ and $x_1$, the procedure

```
FloatDDR1 Poly::Eval(const FloatV &x)
```

performs essentially the same steps as `Poly::Eval_1DD`. (One slight difference is that, in addition to returning the value of the first divided difference, the `FloatDDR1` version also returns the results of evaluating the polynomial on $x_0$ and $x_1$.) □

If we attempt to use `FloatDDR1` arithmetic in a procedure that is not “accumulative” in the independent variable, with only “right-accumulative” quotients, the overload-resolution mechanism of the C++ compiler will detect and report a problem.

Some empirical results that illustrate the advantages of `FloatDDR1` arithmetic in a useful application are presented at the end of Section 8.

As with the methods discussed in Sections 3 and 5, `FloatDDR1` arithmetic can be used to produce values of interest for computational differentiation. For instance, suppose we have transformed procedure `F`:

```
float F(float x);   ⟹   FloatDDR1 F(const FloatV &x);
```

When all of the $x_i$ values in the actual parameter supplied for `FloatV x` are the same value, say $\bar{x}$, then the `FloatDDR1` value returned as the output holds the Taylor coefficients for the expansion of `F` at $\bar{x}$ (*cf.* Formula (5.9)). Thus, the `FloatV` divided-difference arithmetic generalizes previously known techniques for producing accurate Taylor coefficients for functions defined by programs [57,58].

## 7. MULTI-DIMENSIONAL COMPUTATIONAL DIVIDED DIFFERENCING

In this section, we explain how to define a third divided-differencing arithmetic that extends our techniques to handle multi-dimensional computational divided differencing (*i.e.*, computational divided differencing of functions of several variables).

### 7.1. Background Discussion

As background to the material that will be discussed in Section 7.2, let us reiterate a few important points concerning the divided-difference tables that result from computational divided differencing for functions of a single variable. In the following discussion, we assume that the divided-difference table in question has been constructed with respect to some known collection of values $x_0$, $x_1$, ..., $x_n$.

As mentioned at the beginning of Section 6, a divided-difference table can be thought of as a (redundant) representation of an interpolating polynomial. For instance, if you have a divided-difference table $T$ (and know the appropriate vector of values $x_0$, $x_1$, ..., $x_n$, as well), you can explicitly construct the interpolating polynomial in Newton form by using the values in the first row of $T$ in accordance with Eqn. (6.1). One of the consequences of this point is so central to what follows in Section 7.2 that it is worthwhile to state it explicitly and to introduce some helpful notation:

OBSERVATION 7.1 [Representation Principle]. A divided-difference table $T$ is a finite representation of a function $\mathbf{F}[\![T]\!]$ defined by Eqn. (6.1). (Note that if $F = \mathbf{F}[\![T]\!]$, then $T = F^{\triangledown}$.)

Given two divided-difference tables, $T_1$ and $T_2$, that are defined with respect to the same set of points $x_0$, $x_1$, ..., $x_n$, the operations of matrix addition, subtraction, multiplication, and division applied to $T_1$ and $T_2$ yield representations of the sum, difference, product, and quotient, respectively, of $\mathbf{F}[\![T_1]\!]$ and $\mathbf{F}[\![T_2]\!]$. □

It is also worthwhile restating the Computational Divided-Differencing Principle (Observation 5.6), adding the additional remark that is highlighted below in italics:

OBSERVATION 7.2 [Computational Divided-Differencing Principle Redux]. Rather than computing a divided-difference table with respect to the points $x_0$, $x_1$, ..., $x_n$ by invoking the program $n+1$ times and then applying Eqns. (5.1) and (5.2), we may instead evaluate the program (once) using a divided-difference arithmetic that overloads arithmetic operations as matrix operations, substituting $A_{[x_0,\cdots,x_n]}$ for each occurrence of the formal parameter $x$, and $c * I$ for each occurrence of a constant $c$. *Furthermore, this principle can be applied to divided-difference tables for functions on any field (because addition, subtraction, multiplication, and division operations are required, together with additive and multiplicative identity elements).* □

## 7.2. Computational Divided Differencing of Functions of Several Variables

We now consider the problem of defining an appropriate notion of divided differencing for a function $F$ of several variables. Observation 7.1 provides some guidance, as it suggests that the generalized divided-difference table for $F$ that we are trying to create should also be thought of as a representation of a function of several variables that *interpolates* $F$. Such a generalized computational divided-differencing technique will be based on the combination of Observations 7.1 and 7.2.

Because we have already used the term *higher-order* to refer generically to second, third, ..., $n^{th}$ divided differences, we use the term *higher-kind* to refer to the generalized divided-difference tables that arise with functions of several variables. In the remainder of this section, we make use of an alternative notation for the divided-difference operator $\cdot^{\triangledown[x_0,\cdots,x_n]}$:

$$\mathbf{DD}^1{}_{[x_0,x_1,\cdots,x_n]}[\![F]\!] \;=_{df}\; F^{\triangledown[x_0,\cdots,x_n]}.$$

We use $\mathbf{DD}^1[\![F]\!]$ when the $x_i$ are understood. The notation $\mathbf{DD}^1[\![\cdot]\!]$ refers to divided-difference tables of kind 1 (the kind we are already familiar with from Section 5). Below, we use $\mathbf{DD}^2[\![\cdot]\!]$ to refer to divided-difference tables of kind 2; in general, we use $\mathbf{DD}^k[\![\cdot]\!]$ to refer to divided-difference tables of kind $k$.

To understand the basic principle that underlies our approach, consider the problem of creating a surface that interpolates a two-variable function $F(x, y)$ with respect to a grid formed by three coordinate values $x_0$, $x_1$, $x_2$ in the $x$-dimension, and four coordinate values $y_0$, $y_1$, $y_2$, $y_3$ in the $y$-dimension. The clearest way to explain the technique in common programming-language terminology involves currying $F$. That is, instead of working with $F\colon \mathit{float} \times \mathit{float} \to \mathit{float}$, we work with $F\colon \mathit{float} \to \mathit{float} \to \mathit{float}$. We can create (a representation of) an interpolating surface for $F$ by building a divided-difference table (of kind 2) using the functions $F(x_0)$, $F(x_1)$, and $F(x_2)$, each of which is of type $\mathit{float} \to \mathit{float}$, as the “interpolation points”.

Note that this process requires that we be capable of performing addition, subtraction, multiplication, and division of *functions*. However, each of the functions $F(x_0)$, $F(x_1)$, and $F(x_2)$ is itself a one-argument function for which we can create a representation, namely by building the divided-difference tables $\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_0)]\!]$, $\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_1)]\!]$, and $\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_2)]\!]$ (with respect to the coordinate values $y_0$, $y_1$, $y_2$, and $y_3$). By Observation 7.2, the arithmetic operations on functions $F(x_0)$, $F(x_1)$, and $F(x_2)$ needed to create $\mathbf{DD}^2{}_{[x_{0,2}],[y_{0,3}]}[\![F]\!]$ can be carried out by performing matrix operations on the matrices $\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_0)]\!]$, $\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_1)]\!]$, and $\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_2)]\!]$. For instance,

$$\begin{aligned}
\mathbf{DD}^1{}_{[y_{0,3}]}[\![F[x_0, x_1]]\!] &= \mathbf{DD}^1{}_{[y_{0,3}]}[\![\, \frac{F(x_0) - F(x_1)}{x_0 - x_1} \,]\!] \\
&= \frac{\mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_0)]\!] - \mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_1)]\!]}{x_0 - x_1}\,. \qquad (7.3)
\end{aligned}$$

In short, the idea is that a divided-difference table of kind 2 for function $F$ is the matrix of matrices shown below:[9]

$$\mathbf{DD}^2{}_{[x_{0,2}],[y_{0,3}]}[\![F]\!] = \begin{pmatrix} \mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_0)]\!] & \mathbf{DD}^1{}_{[y_{0,3}]}[\![F[x_0, x_1]]\!] & \mathbf{DD}^1{}_{[y_{0,3}]}[\![F[x_0, x_1, x_2]]\!] \\ & \mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_1)]\!] & \mathbf{DD}^1{}_{[y_{0,3}]}[\![F[x_1, x_2]]\!] \\ & & \mathbf{DD}^1{}_{[y_{0,3}]}[\![F(x_2)]\!] \end{pmatrix} \qquad (7.4)$$

$$= \begin{pmatrix}
\begin{pmatrix} F(x_0)(y_0) & F(x_0)[y_{0,1}] & F(x_0)[y_{0,2}] & F(x_0)[y_{0,3}] \\ & F(x_0)(y_1) & F(x_0)[y_{1,2}] & F(x_0)[y_{1,3}] \\ & & F(x_0)(y_2) & F(x_0)[y_{2,3}] \\ & & & F(x_0)(y_3) \end{pmatrix} &
\begin{pmatrix} F[x_{0,1}](y_0) & F[x_{0,1}][y_{0,1}] & F[x_{0,1}][y_{0,2}] & F[x_{0,1}][y_{0,3}] \\ & F[x_{0,1}](y_1) & F[x_{0,1}][y_{1,2}] & F[x_{0,1}][y_{1,3}] \\ & & F[x_{0,1}](y_2) & F[x_{0,1}][y_{2,3}] \\ & & & F[x_{0,1}](y_3) \end{pmatrix} &
\begin{pmatrix} F[x_{0,2}](y_0) & F[x_{0,2}][y_{0,1}] & F[x_{0,2}][y_{0,2}] & F[x_{0,2}][y_{0,3}] \\ & F[x_{0,2}](y_1) & F[x_{0,2}][y_{1,2}] & F[x_{0,2}][y_{1,3}] \\ & & F[x_{0,2}](y_2) & F[x_{0,2}][y_{2,3}] \\ & & & F[x_{0,2}](y_3) \end{pmatrix} \\
\begin{pmatrix} \quad \end{pmatrix} &
\begin{pmatrix} F(x_1)(y_0) & F(x_1)[y_{0,1}] & F(x_1)[y_{0,2}] & F(x_1)[y_{0,3}] \\ & F(x_1)(y_1) & F(x_1)[y_{1,2}] & F(x_1)[y_{1,3}] \\ & & F(x_1)(y_2) & F(x_1)[y_{2,3}] \\ & & & F(x_1)(y_3) \end{pmatrix} &
\begin{pmatrix} F[x_{1,2}](y_0) & F[x_{1,2}][y_{0,1}] & F[x_{1,2}][y_{0,2}] & F[x_{1,2}][y_{0,3}] \\ & F[x_{1,2}](y_1) & F[x_{1,2}][y_{1,2}] & F[x_{1,2}][y_{1,3}] \\ & & F[x_{1,2}](y_2) & F[x_{1,2}][y_{2,3}] \\ & & & F[x_{1,2}](y_3) \end{pmatrix} \\
\begin{pmatrix} \quad \end{pmatrix} & \begin{pmatrix} \quad \end{pmatrix} &
\begin{pmatrix} F(x_2)(y_0) & F(x_2)[y_{0,1}] & F(x_2)[y_{0,2}] & F(x_2)[y_{0,3}] \\ & F(x_2)(y_1) & F(x_2)[y_{1,2}] & F(x_2)[y_{1,3}] \\ & & F(x_2)(y_2) & F(x_2)[y_{2,3}] \\ & & & F(x_2)(y_3) \end{pmatrix}
\end{pmatrix}$$

In what follows, it is convenient to express functions using lambda notation (*i.e.*, in $\lambda z.\, \mathit{exp}$, $z$ is the name of the formal parameter, and *exp* is the function body). For instance, $\lambda x.\, \lambda y.\, x$ denotes the curried two-

[9]This idea is a specific instance of the very general approach to surface approximation via the tensor-product construction given in [14, Chapter XVII].

argument function (of type *float* → *float* → *float*) that merely returns its first argument. For our purposes, the advantage of lambda notation is that it provides a way to express the anonymous one-argument function that is returned when a curried two-argument function is supplied a value for its first argument (*e.g.*, $(\lambda x.\, \lambda y.\, x)(x_0)$ returns $\lambda y.\, x_0$).

It is instructive to consider some concrete instances of $\mathbf{DD}^2_{[x_{0,2}],[y_{0,3}]}[\![F]\!]$ for various $F$'s:

EXAMPLE 7.5. Consider the function $\lambda x.\, \lambda y.\, x$. For $0 \le i \le 2$, we have

$$\mathbf{DD}^1_{[y_{0,3}]}[\![(\lambda x.\, \lambda y.\, x)(x_i)]\!] = \mathbf{DD}^1_{[y_{0,3}]}[\![\lambda y.\, x_i]\!] = \begin{pmatrix} x_i & 0 & 0 & 0 \\ & x_i & 0 & 0 \\ & & x_i & 0 \\ & & & x_i \end{pmatrix}$$

and, for $0 \le i \le 1$, we have

$$\begin{aligned}\mathbf{DD}^1_{[y_{0,3}]}[\![(\lambda x.\, \lambda y.\, x)[x_i, x_{i+1}]]\!] &= \mathbf{DD}^1_{[y_{0,3}]}[\![ \frac{\lambda y.\, x_i - \lambda y.\, x_{i+1}}{x_i - x_{i+1}} ]\!] \\ &= \mathbf{DD}^1_{[y_{0,3}]}[\![\lambda y.\, 1]\!] \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & 0 \\ & & 1 & 0 \\ & & & 1 \end{pmatrix}\end{aligned}$$

Consequently,

$$\mathbf{DD}^2_{[x_{0,2}],[y_{0,3}]}[\![\lambda x.\, \lambda y.\, x]\!] = \begin{pmatrix} \begin{pmatrix} x_0 & 0 & 0 & 0 \\ & x_0 & 0 & 0 \\ & & x_0 & 0 \\ & & & x_0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & 0 \\ & & 1 & 0 \\ & & & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} \\ \begin{pmatrix} \\ \\ \\ \\ \end{pmatrix} & \begin{pmatrix} x_1 & 0 & 0 & 0 \\ & x_1 & 0 & 0 \\ & & x_1 & 0 \\ & & & x_1 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & 0 \\ & & 1 & 0 \\ & & & 1 \end{pmatrix} \\ \begin{pmatrix} \\ \\ \\ \\ \end{pmatrix} & \begin{pmatrix} \\ \\ \\ \\ \end{pmatrix} & \begin{pmatrix} x_2 & 0 & 0 & 0 \\ & x_2 & 0 & 0 \\ & & x_2 & 0 \\ & & & x_2 \end{pmatrix} \end{pmatrix} \tag{7.6}$$

□

EXAMPLE 7.7. Consider the function $\lambda x.\, \lambda y.\, y$. For $0 \le i \le 2$, we have

$$\mathbf{DD}^1_{[y_{0,3}]}[\![(\lambda x.\, \lambda y.\, y)(x_i)]\!] = \mathbf{DD}^1_{[y_{0,3}]}[\![\lambda y.\, y]\!] = \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix}$$

and, for $0 \le i \le 1$, we have

$$\begin{aligned}\mathbf{DD}^1{}_{[y_{0,3}]}[\![(\lambda x.\,\lambda y.\,y)[x_i,x_{i+1}]]\!] &= \mathbf{DD}^1{}_{[y_{0,3}]}[\![\ \frac{\lambda y.\,y-\lambda y.\,y}{x_i-x_{i+1}}\ ]\!] \\ &= \mathbf{DD}^1{}_{[y_{0,3}]}[\![\lambda y.\,0]\!] \\ &= \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix}\end{aligned}$$

Consequently, we have

$$\mathbf{DD}^2{}_{[x_{0,2}],[y_{0,3}]}[\![\lambda x.\,\lambda y.\,y]\!] = \begin{pmatrix} \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} \\ \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} & \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} \\ \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} & \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} & \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix} \end{pmatrix} \tag{7.8}$$

□

Moreover, Observation 7.2 tells us that divided-difference tables for functions of two variables can be built up by means of a divided-difference arithmetic that operates on matrices of matrices. That is, we can build up divided-difference tables of kind 2 for more complex functions of $x$ and $y$ by using operations on matrices of matrices, substituting $\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,x]\!]$ for each occurrence of the formal parameter $x$ in the function, and $\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,y]\!]$ for each occurrence of the formal parameter $y$.

EXAMPLE 7.9. For the function $\lambda x.\,\lambda y.\,x\times y$, $\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,x\times y]\!]$ can be created by *multiplying the matrices* $\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,x]\!]$ and $\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,y]\!]$ from Eqns. (7.6) and (7.8), respectively:

$$\begin{aligned}\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,x\times y]\!] &= \mathbf{DD}^2[\![(\lambda x.\,\lambda y.\,x)\times(\lambda x.\,\lambda y.\,y)]\!] \qquad (7.10) \\ &= \mathbf{DD}^2[\![\lambda x.\,\lambda y.\,x]\!]\times\mathbf{DD}^2[\![\lambda x.\,\lambda y.\,y]\!]\end{aligned}$$

$$= \begin{pmatrix} \begin{pmatrix} x_0y_0 & x_0 & 0 & 0 \\ & x_0y_1 & x_0 & 0 \\ & & x_0y_2 & x_0 \\ & & & x_0y_3 \end{pmatrix} & \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} \\ \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} & \begin{pmatrix} x_1y_0 & x_1 & 0 & 0 \\ & x_1y_1 & x_1 & 0 \\ & & x_1y_2 & x_1 \\ & & & x_1y_3 \end{pmatrix} & \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix} \\ \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} & \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix} & \begin{pmatrix} x_2y_0 & x_2 & 0 & 0 \\ & x_2y_1 & x_2 & 0 \\ & & x_2y_2 & x_2 \\ & & & x_2y_3 \end{pmatrix} \end{pmatrix}$$

Note that the (0,1) entry in the above matrix, namely

$$\begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix}$$

was obtained via the calculation

$$\begin{pmatrix} x_0 & 0 & 0 & 0 \\ & x_0 & 0 & 0 \\ & & x_0 & 0 \\ & & & x_0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & 0 \\ & & 1 & 0 \\ & & & 1 \end{pmatrix} \times \begin{pmatrix} y_0 & 1 & 0 & 0 \\ & y_1 & 1 & 0 \\ & & y_2 & 1 \\ & & & y_3 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 \\ & & 0 & 0 \\ & & & 0 \end{pmatrix} \qquad (7.11)$$

and *not* by the use of Eqn. (7.3), which involves a matrix subtraction, a scalar subtraction, and a scalar division:

$$\mathbf{DD}^1{}_{[y_{0,3}]}[\![(\lambda x.\,\lambda y.\,x \times y)[x_0, x_1]]\!] = \frac{\mathbf{DD}^1{}_{[y_{0,3}]}[\![\lambda y.\,x_0 \times y]\!] - \mathbf{DD}^1{}_{[y_{0,3}]}[\![\lambda y.\,x_1 \times y]\!]}{x_0 - x_1}$$

$$= \frac{\begin{pmatrix} x_0y_0 & x_0 & 0 & 0 \\ & x_0y_1 & x_0 & 0 \\ & & x_0y_2 & x_0 \\ & & & x_0y_3 \end{pmatrix} - \begin{pmatrix} x_1y_0 & x_1 & 0 & 0 \\ & x_1y_1 & x_1 & 0 \\ & & x_1y_2 & x_1 \\ & & & x_1y_3 \end{pmatrix}}{x_0 - x_1} \qquad (7.12)$$

Expressions (7.11) and (7.12) are equivalent over real numbers, but not over floating-point numbers. By sidestepping the explicit subtraction and division operations in expression (7.12), expression (7.11) avoids the potentially catastrophic cancellation that can occur with floating-point arithmetic. □

The principle illustrated in Example 7.9 gives us the machinery that we need to perform computational divided differencing for bivariate functions defined by programs. As usual, computational divided differencing is performed by changing the types of formal parameters, local variables, and return values to the type of an appropriate divided-difference arithmetic.

Furthermore, these ideas can be applied to a function $F$ with an arbitrary number of variables: when $F$ has $k$ variables, $\mathbf{DD}^k[\![F]\!]$, $F$'s divided-difference table of kind $k$, is a matrix of matrices of ... of matrices nested to depth $k$. Currying with respect to the first parameter of $F$ "peels off" one dimension; $\mathbf{DD}^k[\![F]\!]$ is a matrix whose entries are divided-difference tables of kind $k-1$ (*i.e.*, matrices of matrices of ... of matrices nested to depth $k-1$). For instance, the diagonal entries are the divided-difference tables of kind $k-1$ for the $(k-1)$-parameter functions $F(x_0)$, $F(x_1)$, ..., $F(x_n)$ (*i.e.*, $\mathbf{DD}^{k-1}[\![F(x_0)]\!]$, $\mathbf{DD}^{k-1}[\![F(x_1)]\!]$, ..., $\mathbf{DD}^{k-1}[\![F(x_n)]\!]$).

To implement this approach in C++, we define two classes and one class template:

- Class template `template <int k> class DivDiffArith` can be instantiated with a value `k` $> 0$ to represent divided-difference tables of kind `k`. Each object of class `DivDiffArith<k>` has links to sub-objects of class `DivDiffArith<k−1>`.
- Class `DivDiffArith<0>` represents the base case; `DivDiffArith<0>` objects simply hold a single `float`.
- Class `IntVector`, a vector of `int`'s, is used to describe the number of points in each dimension of the grid of coordinate points.

Excerpts from the definitions of these classes are shown below:

```
template <int k> class DivDiffArith {
 public:
  int numPts;
  DivDiffArith<k−1> **divDiffTable;  // Two-dimensional upper-triangular array

  DivDiffArith(const FloatV &v, const IntVector &grid, int d);
  DivDiffArith(float, const IntVector &grid);        // constant; shape conforms to grid
  DivDiffArith(float, const DivDiffArith<k−1> &dda); // constant; shape conforms to dda

  DivDiffArith<k>& operator+ (const DivDiffArith<k> &) const; // binary addition
  DivDiffArith<k>& operator− (const DivDiffArith<k> &) const; // binary subtraction
  DivDiffArith<k>& operator* (const DivDiffArith<k> &) const; // binary multiplication
  DivDiffArith<k>& operator/ (const DivDiffArith<k> &) const; // binary division
};
```

```
class DivDiffArith<0> {
 public:
  float value;

  DivDiffArith(float v = 0.0);         // Default constructor
  DivDiffArith(const FloatV &v, const IntVector &grid, int d);

  DivDiffArith<0>& operator+ (const DivDiffArith<0> &) const; // binary addition
  DivDiffArith<0>& operator− (const DivDiffArith<0> &) const; // binary subtraction
  DivDiffArith<0>& operator* (const DivDiffArith<0> &) const; // binary multiplication
  DivDiffArith<0>& operator/ (const DivDiffArith<0> &) const; // binary division
};
```

```
class IntVector {
 public:
  int numPts;
  int *val;   // An array of values: val[0]..val[numPts−1]

  IntVector();
  IntVector(int N, ...);               // Construct IntVector given N values

  IntVector& operator<< (const int i); // left shift
};
```

The operations of class `DivDiffArith<k>` are overloaded in a fashion similar to those of class `FloatDD`. (Class `FloatDD` is essentially identical to `DivDiffArith<1>`.) For instance, the overloaded multiplication operator performs matrix multiplication:

```
template <int k>
DivDiffArith<k>& DivDiffArith<k>::operator* (const DivDiffArith<k> &dda) const
{
  assert(numPts == dda.numPts);

  DivDiffArith<k> *ans = new DivDiffArith<k>(numPts);
  for (int r = 0; r < numPts; r++) {
    for (int c = r; c < numPts; c++) {
      DivDiffArith<k−1> temp((float)0.0, divDiffTable[r][c]);  // temp = 0.0
      for (int j = r; j <= c; j++) {
        temp += divDiffTable[r][j] * dda.divDiffTable[j][c];
      }
      ans−>divDiffTable[r][c] = temp;
    }
  }
  return *ans;
}
```

Class `DivDiffArith<k>` has two constructors for creating a `DivDiffArith<k>` object from a `float` constant. They differ only in their second arguments (an `IntVector` versus a `DivDiffArith<k>`), which are used to determine the appropriate dimensions to use at each level in the nesting of matrices.

Suppose variable `z` is the independent variable associated with argument position `d+1`. To generate a `DivDiffArith<k>` object for a given set of grid values for `z` (say $z_0$, ..., $z_m$), a `FloatV` with the values of the $z_i$ is created, and then passed to the following `DivDiffArith<k>` constructor:

```
template <int k>
DivDiffArith<k>::DivDiffArith(const FloatV &v, const IntVector &grid, int d) :
  numPts(grid.val[0]),
  divDiffTable(calloc_ut< k >(numPts))
{
  assert(grid.val[d] == v.numPts);

  IntVector tail = grid << 1;
  for (int r = 0; r < numPts; r++) {
    for (int c = r; c < numPts; c++) {
      divDiffTable[r][c] = DivDiffArith<k−1>((float)0.0,tail);
    }
  }
  if (d == 0) {
    DivDiffArith<k−1> one((float)1.0, tail);
    for (int i = 0; i < numPts; i++) {
      divDiffTable[i][i] = DivDiffArith<k−1>(v.val[i],tail);
      if (i < numPts − 1) {
        divDiffTable[i][i+1] = one;
      }
    }
  }
  else {
    for (int i = 0; i < numPts; i++) {
      divDiffTable[i][i] = DivDiffArith<k−1>(v,tail,d-1);
    }
  }
}
```

EXAMPLE 7.13. The following code fragment generates two `DivDiffArith<2>` values, `x` and `y`, which correspond to the matrices shown in Eqns. (7.6) and (7.8), respectively:

```
IntVector grid(2,3,4);
FloatV fv_x(3,x₀,x₁,x₂);
DivDiffArith<2> x(fv_x,grid,0);  // argument position 1
FloatV fv_y(4,y₀,y₁,y₂,y₃);
DivDiffArith<2> y(fv_y,grid,1);  // argument position 2
```

□

EXAMPLE 7.14. Consider a C++ class `BivariatePoly` that represents bivariate polynomials, and a member function `BivariatePoly::Eval` that evaluates a polynomial via a bivariate version of Horner’s rule:

```
class BivariatePoly {
 public:
  float Eval(float,float);
 private:
  int degree1,degree2;
  // Array coeff[0..degree1][0..degree2]
  float **coeff;
};
```

```
// Evaluation via bivariate Horner’s rule
float BivariatePoly::Eval(float x,float y){
  float ans = 0.0;
  for (int i = degree1; i >= 0; i−−){
    float temp = 0.0;
    for (int j = degree2; j >= 0; j−−){
      temp = temp * y + coeff[i][j];
    }
    ans = ans * x + temp;
  }
  return ans;
}
```

Similar to what has been done in Examples 3.4, 4.3, 5.8, and 6.3, computational divided differencing is carried out on this version of `Eval` by changing the types of its formal parameters, local variables, and return value from `float` to `DivDiffArith<2>`.

```
// Evaluation via bivariate Horner’s rule
DivDiffArith<2> BivariatePoly::Eval(const DivDiffArith<2> &x, const DivDiffArith<2> &y)
{
  DivDiffArith<2> ans(0.0,x);       // ans = 0.0
  for (int i = degree1; i >= 0; i−−){
    DivDiffArith<2> temp(0.0,y);    // temp = 0.0
    for (int j = degree2; j >= 0; j−−){
      temp = temp * y + coeff[i][j];
    }
    ans = ans * x + temp;
  }
  return ans;
}
```

To use this procedure to create a divided-difference table of kind 2 for a given variable `P` of type `BivariatePoly*`, with respect to the 3-by-4 grid $\{x_0,x_1,x_2\} \times \{y_0,y_1,y_2,y_3\}$, we would generate the `IntVector grid` and `DivDiffArith<2>` values `x` and `y` as shown in Example 7.13, and then invoke

```
P−>Eval(x,y);
```

□

One final point concerning this approach: it is worthwhile noting that by generalizing the grid descriptors slightly, it is possible to devise an even more general divided-differencing arithmetic that is *heterogeneous* in shape with respect to different argument positions. By “heterogeneous”, we mean that full two-dimensional (upper-triangular) divided-difference tables could be provided for some argument positions, while other argument positions could just provide a single row of divided differences (*i.e.*, one-dimensional, `FloatDDR1`-like tables). By this means, when a procedure body is “accumulative” in certain of its formal parameters but not others, it would be possible to tailor the divided-differencing version of the procedure to improve its efficiency. (In the case of procedure

`DivDiffArith<2> BivariatePoly::Eval`,

it would be possible to specify that both argument positions provide `FloatDDR1`-like tables.)

## 8. PAIGE’S WORK ON FINITE DIFFERENCING OF COMPUTABLE EXPRESSIONS

In earlier sections, and also in Section 9, we have attempted to place the ideas that are developed in this paper in their proper context by describing how they relate to previous work on computational-differentiation [68,56,26,5,27] and to previous work on the creation of accurate divided-difference tables for expressions [50,46]. In the present section, we discuss some additional intellectual forbearers of our work, focusing particularly on how our ideas relate to Robert Paige’s work on finite differencing of set-valued expressions in SETL programs.

Starting in the mid-70s, Paige studied how finite-differencing transformations of applicative set-former expressions could be exploited to optimize loops in very-high-level languages, such as SETL. References [52,53] are just two of many works that Paige wrote about this subject, and these ideas were implemented in his RAPTS system [53,54]. Some of the techniques that Paige explored have their roots in earlier work by Earley [15,16]. Independently of and contemporaneously with Paige, similar loop-optimization methods targeted toward very-high-level set-theoretic languages were investigated by Fong and Ullman [18,19,20]. More recently, Liu and Stoller have used some extensions of these ideas to optimize array computations [42] and recursive programs [44]. Liu et al. have also shown how such transformations can be applied to derive algorithms for incremental-computation problems (*i.e.*, problems in which the goal is to maintain the value of some function $F(x)$ as the input $x$ undergoes small changes) [38,39,40,41,43].

Both Paige [52] and Paige and Koenig [53] provide lengthy discussions of the roots of the ideas that are developed in those papers. The basic idea for optimizing SETL loops is described as a generalization of strength reduction, an idea attributed to John Cocke from the 60s, whereby a loop is transformed so that a multiplication operation in the loop is eliminated in favor of an addition, as shown below:

```
                             i = ...;
                             T = i*c;          // T depends on i
i = ...;                     deltaT = delta*c;
while (...) {          ⟹     while (...) {
  ... i*c ...;                 ... T ...;      // T replaces i*c
  i = i + delta;               i = i + delta;  // change to i
}                              T = T + deltaT  // update of T
                             }
```

This transformation improves the running time of the loop if the cost of the additions performed by the transformed loop are less than the cost of the multiplications performed in the original loop. In [12], Cocke and Schwartz presented a variety of strength-reduction transformations for use in optimizing compilers.

Paige’s work on loop optimization in SETL was based on the observation that a similar transformation could be applied to loops that involve set-former expressions. In this transformation, an expensive set-former expression in a loop is replaced by a set-initialization statement (placed before the loop) together with a set-update operation (placed inside the loop):

```
                                        A = ...;
A = ...;                                T = { x ∊ A | x%2 == 0 }; // T depends on A
while (...) {                           while (...) {
  ... { x ∊ A | x%2 == 0 } ...; ⟹        ... T ...;   // T replaces the set former
  d = ...;                                d = ...;
  A = A ∪ d;                              A = A ∪ d;                // change to A
}                                         if (d%2 == 0) T = T ∪ d;  // update of T
                                        }
```

In the transformed program shown above on the right, the expression `{ x ∊ A | x%2 == 0 }` in the loop is replaced by a use of `T`. Because the statement

```
A = A ∪ d;
```

may alter the value of variable `A`, just after this statement a new statement is introduced:

```
if (d%2 == 0) T = T ∪ d;
```

The latter statement updates the value of variable `T` to have the same value that the expression `{ x ∊ A | x%2 == 0 }` has when evaluated with the new value of variable `A`.[10]

Of more direct relevance to the topic of the present paper is the discussion in Paige's papers in which he points out affinities between his work, on the one hand, and numerical differentiation and numerical finite-difference methods, on the other hand. For instance, Paige and Koenig describe the relationship between their SETL finite-differencing methods and numerical finite-difference methods as follows [53, pp. 403−404]:

> It is interesting to note that the origins of our method may be traced back to the finite difference techniques introduced by the English mathematician Henry Briggs in the sixteenth century. His method, which can be used to generate a sequence of polynomial values $p(x_0)$, $p(x_0+h)$, $p(x_0+2h)$, . . ., hinges on the following idea. For a given polynomial $p(x)$ of degree $n$ and an increment $h$, the first difference polynomial
>
> $$p_1(x) = p(x+h) - p(x)$$
>
> is of degree $n-1$ or less, the second difference polynomial
>
> $$p_2(x) = p_1(x+h) - p_1(x)$$
>
> is of degree $n-2$ or less, . . ., and, finally, $p_n(x)$ must be a constant. Thus, to tabulate successive values of $p(x)$ starting with $x = x_0$, we can perform these two steps:
>
> 1. Calculate initial values for $p(x_0)$, $p_1(x_0)$, . . ., $p_n(x_0)$ and store them in $t(1)$, $t(2)$, . . ., $t(n+1)$.
> 2. Generate the desired polynomial table by iterating over the following code block:

[10]The code fragment `if (d%2 == 0) T = T ∪ d;` is called a *post-derivative* of `T = { x ∊ A | x%2 == 0 };` with respect to the change `A = A ∪ d;`. Similarly, a code fragment for updating variable `T` that is placed before the change `A = A ∪ d;` is called a *pre-derivative*.

```
print x, t(1);                     print x and p(x)
t(1) := t(1) + t(2);               $ place new values for
t(2) := t(2) + t(3);               $ p(x), p_1(x),
            .                      $ ..., p_{n-1}(x) into
            .
            .
t(n) := t(n) + t(n-1);             $ t(1), t(2), ..., t(n).
x := x + h;                        $
```

Note that Briggs's method requires only $n$ additions in step 2 to compute each new polynomial value, while Horner's rule to compute a fresh polynomial value costs $n$ additions and $n$ multiplications.

They relate Briggs's method to strength reduction in the following passage [53, pp. 404−405]:

Although Cocke's technique does not treat polynomials as special objects, strength reduction is sufficiently powerful to transform a program involving repeated calculations of a polynomial according to Horner's rule into an equivalent program that essentially uses the more efficient finite difference method of Briggs. Indeed, this is a surprising and important result that demonstrates that the success of polynomial evaluation by differencing results from properties of the elementary operations used to form polynomials rather than from properties exclusive to polynomials. In other words, Cocke's method works because the following distributive and associative laws hold for sums and products:

$$(i \pm delta) * c \Rightarrow i * c \pm delta * c;$$
$$(i \pm delta) + c \Rightarrow (i + c) \pm delta.$$

In [12] Cocke and Schwartz extend this idea to show how reduction in strength (which we call finite differencing) applies to a wide range of arithmetic operations that exhibit appropriate distributive properties.

Later in the paper, after Paige and Koenig have introduced their rules for finite differencing of set-former expressions with respect to changes in argument values (an operation that they sometimes call "differentiation"), they return to the discussion of Briggs's method [53, pp. 421]:

Profitable differentiation of an expression $f$ can sometimes be supported by differentiating $f$ together with a chain of auxiliary expressions (as in Briggs's first, second, ..., difference polynomials . . . ). Thus, the prederivative $\nabla^{-}E \langle x \mathrel{+:=} delta; \rangle$ of the $n^{th}$ degree polynomial $E = P(x)$ is

$$E \mathrel{+:=} P_1(x)$$

where $P_1(x)$ is the first difference polynomial. However, for the prederivative code above to be inexpensive, we must also differentiate the second, third, ..., $n^{th}$ difference polynomials, denoted $E_i = P_i(x)$, $i = 2 \,..\, n$. To realize Briggs's efficient technique, we consider the extended prederivative (of expressions ordered carefully into a "differentiable chain") $\nabla^{-}E_{n-1}, \ldots, E_1, E \langle x \mathrel{+:=} delta; \rangle$ that expands into

$$\begin{array}{l} E \mathrel{+:=} E_1; \\ E_1 \mathrel{+:=} E_2; \\ \quad\quad \vdots \\ E_{n-1} \mathrel{+:=} E_n; \end{array}$$

Essentially all of the material that Paige and Koenig present in their paper to relate their work to Briggs's method has been quoted above. However, a few details about the derivation of Briggs's method were not spelled out in their treatment, which we now attempt to rectify. We will show below that computational divided differencing supplies a clean way to handle an important step in the derivation about which Paige and Koenig are silent.

The initial program for tabulating a polynomial at a collection of equally spaced points can be written in C++ as follows:[11]

```
void Poly::Tabulate(float start, int numPoints, float h)
{
  float x = start;
  float y;
  for (int i = 1; i <= numPoints; i++) {  // Tabulation loop
    y = Eval(x);
    cout << x << ": " << y << endl;
    x += h;
  }
}
```

In the program shown above, `Eval` is the member function of class `Poly` that evaluates a polynomial via Horner's rule (see Section 3), of type

`float Poly::Eval(float x);`

The intention of Paige and Koenig is to transform procedure `Poly::Tabulate` into something like the following version:

```
void Poly::Tabulate(float start, int numPoints, float h)
{
  float x = start;
  float y;
  float E = Eval(start);  // Call on Eval hoisted out of the loop
  float E_1 = ???;        // Unspecified
         .                // initialization
         .                //       "
         .                //       "
  float E_n-1 = ???;      //       "
  float E_n = ???;        //       "
  for (int i = 1; i <= numPoints; i++) {  // Tabulation loop
    y = E;
    cout << x << ": " << y << endl;
    E += E_1;    // Extended pre-derivative
    E_1 += E_2;  // w.r.t. x += h;
          .    //         "
          .    //         "
          .    //         "
    E_n-1 += E_n;  //     "
    x += h;
  }
}
```

However, as indicated by the question marks in the above code, Paige and Koenig do not state explicitly

[11]It should be pointed out that in practice it is better to code the statement in the loop that changes the value of `x` as "`x = start + i * h;`", rather than as "`x += h;`", so that small errors in `h` do not accumulate in `x` due to repeated addition. We have chosen to use the latter form to emphasize the similarities between `Tabulate` and the two earlier strength-reduction examples.

how they plan to arrive at the proper initialization code that is to be placed just before the loop in the transformed program. It is unclear whether they intend to generate the values $E_1$, $E_2$, ..., $E_{n-1}$, $E_n$ by evaluating polynomial P at `start`, `start+h`, ..., `start+(n−1)*h`, `start+n*h` and then create the $E_i$ via subtraction operations, or whether they intend to generate the finite-difference polynomials $P_1$, $P_2$, ..., $P_{n-1}$, $P_n$ symbolically and then apply each of them to `start`. The former method can lead to very inaccurate results (see below), whereas the latter method requires that a substantial amount of symbolic manipulation be performed to generate the $P_i$.

However, the divided-difference arithmetic `FloatDDR1` gives us an easy way to create suitable initialization code that produces accurate values for the $E_i$. This initialization code consists of three steps:

(1) Create a `FloatV` of equally spaced points, starting at `start` and separated by `h`, where the number of points is one more than the degree of the polynomial.

(2) Introduce a single call on the member function

```
FloatDDR1 Poly::Eval(const FloatV &x);
```

to create the first row of the divided-difference table for the polynomial with respect to the given `FloatV`.

(3) Convert the resulting `FloatDDR1` (which holds *divided-difference* values) into the first row of a *finite-difference* table for the polynomial by multiplying each entry by an appropriate adjustment factor (see [13, Lemma 4.1]).

This initialization method is used in the version of `Tabulate` shown below. (In this version of `Tabulate`, the $E_i$ are renamed `diffTable[i]`.)

```
void Poly::Tabulate(float start, int numPoints, float h)
{
  float x = start;
  float y;

  // Create accurate divided-difference table
  FloatV fv(x, degree+1, h);
  FloatDDR1 fddr1 = Eval(fv);   // Calls FloatDDR1 Poly::Eval(const FloatV &);

  // Convert divided-difference entries to finite-difference entries
  float *diffTable(new float[degree+1]);
  float adjustment = 1.0;
  for (int i = 0; i <= degree; i++) {
      diffTable[i] = fddr1.divDiffTable[i] * adjustment;
      adjustment *= (h * (i+1));
  }

  for (int i = 1; i <= numPoints; i++) {  // Tabulation loop
    y = diffTable[0];
    cout << x << ": " << y << endl;
    for (int j = 0; j < degree; j++) {  // Pre-derivative w.r.t. x += h;
      diffTable[j] += diffTable[j+1];
    }
    x += h;
  }
}
```

## Empirical Results: Tabulation of a Polynomial via Briggs's Method

We now present some empirical results that illustrate the advantages of the final version of `Poly::Tabulate` presented above. Again, we work with the polynomial $P(x) = 2.1 * x^3 - 1.4 * x^2 - .6 * x + 1.1$, and perform computations using single-precision floating-point arithmetic.

The final version of `Poly::Tabulate` uses the divided-difference arithmetic `FloatDDR1` in the initialization step that creates the initial finite-difference vector `diffTable`. An alternative way to generate `diffTable` is to evaluate polynomial `P` at `start`, `start+h`, ..., `start+(n−1)*h`, `start+n*h` and then create `diffTable` via subtraction operations, according to the standard definition [13, pp. 214]. However, the latter way of generating `diffTable` involves subtraction operations, and hence is subject to possible catastrophic cancellations. In contrast, the method using computational divided differencing yields a way to create a more accurate initial finite-difference table.

To give a concrete illustration of the benefits, the following table shows what happens when $P(x)$ is evaluated at the 10,001 points in the interval $[0.0, 1.0]$ with a grid spacing of .0001:

| | Evaluate via Horner | Comp. Div. Diff. + Briggs | Standard Finite Diff. + Briggs |
|---|---|---|---|
| $x$ | $P(x)$ | $P(x)$ | $P(x)$ |
| 0.0000 | 1.10000 | 1.10000 | 1.10000 |
| 0.0001 | 1.09994 | 1.09994 | 1.09994 |
| 0.0002 | 1.09988 | 1.09988 | 1.09988 |
| . . . | . . . | . . . | . . . |
| 0.9998 | 1.19942 | 1.19941 | 19844.3 |
| 0.9999 | 1.19971 | 1.19970 | 19850.3 |
| 1.0000 | 1.20000 | 1.19999 | 19856.2 |
| Time (in seconds for 1000 trials) | 7.64 | 5.62 | 5.49 |

The numbers that appear in the rightmost column for $P(.9998)$, $P(.9999)$, and $P(1.0000)$ are *not* typographical errors. What happens is that the standard method for computing the initial finite-difference table involves a catastrophic cancellation, and after 10,000 iterations of the Briggs calculation, the accumulated round-off error makes the values produced diverge widely from the correct answers.

Overall, the method based on computational divided differencing is far more accurate than the one in which the vector needed for Briggs's method is obtained by subtraction operations (and only 2% slower). Furthermore, the results from the method based on computational divided differencing are nearly as accurate as those obtained by reevaluating the polynomial at each point, but the reevaluation method is 36% slower.

## 9. OTHER RELATED WORK

Section 8 described how our ideas relate to Robert Paige's work on finite differencing of set-valued expressions in SETL programs. This section concerns other related work, which falls into four categories:

### Computational Differentiation

Computational differentiation is a well-established area of numerical analysis, with its own substantial literature [68,56,26,5,27]. The importance of the subject is underscored by the fact that the 1995 Wilkinson Prize for outstanding contributions to the field of numerical software was awarded to Chris Bischof (then at Argonne) and Alan Carle (Rice) for the development of the FORTRAN computational-differentiation system ADIFOR 2 [8]. As discussed in Section 4, computational divided differencing is a *generalization* of computational differentiation: a program resulting from computational divided differencing can be used to obtain derivatives (as well as divided differences), whereas a program resulting from computational differentiation can only produce derivatives (and not divided differences).

### Other Work on Accurate Divided Differencing

The program-transformation techniques for creating accurate divided differences described in this paper are based on a 1964 result of Opitz's [50], which was later rediscovered in 1980 by McCurdy [46] and again in 1998 by one of us (Reps). However, Opitz and McCurdy both discuss how to create accurate divided differences only for *expressions*. In this paper, the idea has been applied to the creation of accurate divided differences for functions defined by *programs*.

McCurdy, and later Kahan and Fateman [35] and Rall and Reps [62], have looked at ways to compute accurate divided differences for library functions (*i.e.*, sin, cos, exp, *etc*.).

Kahan and Fateman have also investigated how similar techniques can be used to avoid catastrophic cancellation when evaluating formulas returned by symbolic-algebra systems. In particular, their work was motivated by the observation that naive evaluation of a definite integral $\int_a^b f(x)\,dx$ can sometimes produce meaningless answers: When a symbolic-algebra system produces a closed-form solution for the indefinite integral $\int f(x)\,dx$, say $G(x)$, the result of the computation $G(b) - G(a)$ may have no significant digits due to catastrophic cancellation. Kahan and Fateman show that divided differences can be used to develop accurate numerical formulas that sidestep this problem.

One of the techniques developed by McCurdy for computing accurate divided-difference tables involved first computing just the first row of the table and then generating the rest of the entries by a backfilling algorithm. He studied the conditions under which this technique maintained sufficient accuracy. However, his algorithm for accurately computing the first row of the divided-difference table was based on a series expansion of the function, rather than a divided-difference arithmetic, such as the `FloatDDR1` arithmetic developed in Section 6.

## Other Work on Controlling Round-Off Error in Numerical Computations

Computational differentiation and computational divided differencing are methods for controlling the round-off errors that can arise in two types of numerical computations. An extensive collection of methods for controlling round-off error for a wide variety of numerical computations has been developed by Kulisch's group at the University of Karlsruhe [28,29].

In future work, we plan to investigate the use of such techniques to achieve greater accuracy (and to validate results) in steps of the computational-divided-differencing transformations where operations are implemented by solving systems of linear equations (*cf.* box (5.7)).

## Other Work that Exploits Operator Overloading

The operator-overloading feature of C++ provides a convenient mechanism for implementing the divided-difference arithmetics that are described in this paper. Compared to other methods of providing polymorphism in programming languages, such as *parametric polymorphism* [47] and *inheritance polymorphism* [11], operator overloading has always been something of a poor cousin. In the programming-languages community, operator overloading is sometimes referred to as "*ad hoc* polymorphism", a term that has pejorative overtones. Nevertheless, in addition to its application in computational differentiation and computational divided differencing, operator overloading provides a convenient implementation mechanism for a wide variety of other interesting applications, including partial evaluation [1], safe pointers [2], and expression templates [66].

ACKNOWLEDGEMENTS

We are grateful for discussions with Chris Bischof, Carl de Boor, James O'Brien, and Richard Zippel about various aspects of the work described in the paper.

## REFERENCES

1. Andersen, L.O., "Program analysis and specialization for the C programming language," Ph.D. diss. and Report TOPPS D-203, Datalogisk Institut, Univ. of Copenhagen, Copenhagen, Denmark (May 1994).
2. Austin, T.M., Breach, S.E., and Sohi, G., "Efficient detection of all pointer and array access errors," *Proc. of the ACM SIGPLAN 94 Conf. on Prog. Lang. Design and Implementation,* (Orlando, FL, June 20-24, 1994), *SIGPLAN Not.* **29**(6) pp. 290-301 (June 1994).
3. Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs," pp. 384-396 in *Conf. Rec. of the Twentieth ACM Symp. on Princ. of Prog. Lang.,* (Charleston, SC, Jan. 10-13, 1993), ACM, New York, NY (1993).
4. Beck, T. and Fischer, H., "The if-problem in automatic differentiation," *J. Comp. and Appl. Math.* **50** pp. 119-131 (1994).
5. Berz, M., Bischof, C., Corliss, G.F., and Griewank, A. (eds.), *Computational Differentiation: Techniques, Applications, and Tools,* Soc. for Indust. and Appl. Math., Philadelphia, PA (1996).
6. Binkley, D., "Using semantic differencing to reduce the cost of regression testing," pp. 41-50 in *Proc. of the IEEE Conf. on Softw. Maint.* (Orlando, FL, Nov. 9-12, 1992), IEEE Comp. Soc., Wash., DC (1992).
7. Bischof, C., Carle, A., Corliss, G.F., and Griewank, A., "ADIFOR: Automatic differentiation in a source translator environment," pp. 294-302 in *Proc. of ISSAC 1992: Int. Symp. on Symb. and Alg. Comp.,* (Berkeley, CA, July 27-29, 1992), ACM, New York, NY (1992).
8. Bischof, C., Carle, A., Khademi, P., and Mauer, A., "ADIFOR 2.0: Automatic differentiation of Fortran 77 programs," *IEEE Comp. Sci. and Eng.* **3** pp. 18-32 (1996).
9. Bischof, C., Roh, L., and Mauer, A., "ADIC: An extensible automatic differentiation tool for ANSI-C," Tech. Rep. ANL/MCS-P626-1196, Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL (1997).
10. Bjϕrner, D., Ershov, A.P., and Jones, N.D. (eds.), *Partial Evaluation and Mixed Computation: Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation,* (Gammel Avernaes, Denmark, Oct. 18-24, 1987), North-Holland, New York, NY (1988).
11. Cardelli, L., "A semantics of multiple inheritance," pp. 51-67 in *Semantics of Data Types: Proc. of the Int. Symp.,* (Sophia-Antipolis, France, June 27-29, 1984), *Lec. Notes in Comp. Sci.,* Vol. 173, ed. G. Kahn, D. MacQueen, and G. Plotkin, Springer-Verlag, New York, NY (1984).
12. Cocke, J. and Schwartz, J.T., *Programming Languages and Their Compilers: Preliminary Notes, 2nd Rev. Version*, Courant Inst. of Math. Sci., New York Univ., New York, NY (1970).
13. Conte, S.D. and de Boor, C., *Elementary Numerical Analysis: An Algorithmic Approach,* 2nd. Ed., McGraw-Hill, New York, NY (1972).
14. de Boor, C., *A Practical Guide to Splines,* (Appl. Math. Sciences, Vol. 27), Springer-Verlag, New York, NY (1978).
15. Earley, J., "High-level operations in automatic programming," *Proc. of the ACM SIGPLAN Symp. on Very High Level Languages,* (Mar. 1974), *SIGPLAN Not.* **9**(4)(Apr. 1974).
16. Earley, J., "High-level iterators and a method for automatically designing data structure representation," *J. Comp. Lang.* **1**(4) pp. 321-342 (1976).
17. Fischer, H., "Special problems in automatic differentiation," pp. 43-50 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G.F. Corliss, Soc. for Indust. and Appl. Math., Philadelphia, PA (1992).
18. Fong, A. and Ullman, J., "Induction variables in very high level languages," pp. 104-112 in *Conf. Rec. of the Third ACM Symp. on Princ. of Prog. Lang.,* (Atlanta, GA, Jan. 19-21, 1976), ACM, New York, NY (1976).
19. Fong, A., "Elimination of common subexpressions in very high level languages," pp. 48-57 in *Conf. Rec. of the Fourth ACM Symp. on Princ. of Prog. Lang.,* (Los Angeles, CA, Jan. 17-19, 1977), ACM, New York, NY (1977).
20. Fong, A., "Inductively computable constructs in very high level languages," pp. 21-28 in *Conf. Rec. of the Sixth ACM Symp. on Princ. of Prog. Lang.,* (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).
21. Futamura, Y., "Partial evaluation of computation process − an approach to a compiler-compiler," *Systems, Computers, Controls* **2**(5) pp. 45-50 (1971).
22. Gallagher, K.B. and Lyle, J.R., "Using program slicing in software maintenance," *IEEE Trans. on Softw. Eng.* **SE-17**(8) pp. 751-761 (Aug. 1991).
23. Goldstine, H.H., *A History of Numerical Analysis,* Springer-Verlag, New York, NY (1977).
24. Griewank, A., "On automatic differentiation," pp. 83-108 in *Mathematical Programming: Recent Developments and Applications*, ed. M. Iri and K. Tanabe, Kluwer Academic Publishers, Boston, MA (1989).
25. Griewank, A., "The chain rule revisited in scientific computing," *SIAM News* **24**(May & July, 1991).
26. Griewank, A. and Corliss, G.F. (eds.), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application,* Soc. for Indust. and Appl. Math., Philadelphia, PA (1992).

27. Griewank, A., *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation,* Soc. for Indust. and Appl. Math., Philadelphia, PA (2000).
28. Hammer, R., Hocks, M., Kulisch, U., and Ratz, D., *Numerical Toolbox for Verified Computing I: Basic Numerical Problems,* (Springer Ser. in Comp. Math., Vol. 21), Springer-Verlag, New York, NY (1993).
29. Hammer, R., Hocks, M., Kulisch, U., and Ratz, D., *C++ Toolbox for Verified Computing I: Basic Numerical Problems,* Springer-Verlag, New York, NY (1995).
30. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).
31. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (Jan. 1990).
32. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proc. of the ACM SIGPLAN 90 Conf. on Prog. Lang. Design and Implementation,* (White Plains, NY, June 20-22, 1990), *SIGPLAN Not.* **25**(6) pp. 234-245 (June 1990).
33. Iri, M., "Simultaneous computation of functions, partial derivatives and estimates of rounding errors: Complexity and practicality," *Japan J. Appl. Math.* **1**(2) pp. 223-252 (1984).
34. Jones, N.D., Gomard, C.K., and Sestoft, P., *Partial Evaluation and Automatic Program Generation,* Prentice-Hall Int., Englewood Cliffs, NJ (1993).
35. Kahan, W. and Fateman, R.J., "Symbolic computation of divided differences," Unpublished report, Dept. of Elec. Eng. and Comp. Sci., Univ. of Calif.−Berkeley, Berkeley, CA (1985). (Available at http://www.cs.berkeley.edu/˜fateman/papers/divdiff.pdf.)
36. Kearfott, R.B., "Automatic differentiation of conditional branches in an operator overloading context," pp. 75-81 in *Computational Differentiation: Techniques, Applications, and Tools*, ed. M. Berz, C. Bischof, G.F. Corliss, and A. Griewank, Soc. for Indust. and Appl. Math., Philadelphia, PA (1996).
37. Linnainmaa, S., "Taylor expansion of the accumulated rounding error," *BIT* **16**(1) pp. 146-160 (1976).
38. Liu, Y.A., "Incremental computation: A semantics-based systematic transformation approach," Ph.D. diss. and Tech. Rep. 95-1551, Dept. of Comp. Sci., Cornell Univ., Ithaca, NY (Oct. 1995).
39. Liu, Y.A. and Teitelbaum, T., "Caching intermediate results for program improvement," in *Proc. of the ACM SIGPLAN Symp. on Part. Eval. and Sem.-Based Prog. Manip. (PEPM 95),* (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).
40. Liu, Y.A. and Teitelbaum, T., "Systematic derivation of incremental programs," *Science of Computer Programming* **24** pp. 1-39 (1995).
41. Liu, Y.A., Stoller, S.D., and Teitelbaum, T., "Discovering auxiliary information for incremental computation," pp. 157-170 in *Conf. Rec. of the Twenty-Third ACM Symp. on Princ. of Prog. Lang.,* (St. Petersburg, FL, Jan. 22-24, 1996), ACM, New York, NY (1996).
42. Liu, Y.A. and Stoller, S.D., "Loop optimization for aggregate array computations," in *Proc. of the IEEE 1998 Int. Conf. on Comp. Lang.,* (Chicago, IL, May 1998), IEEE Comp. Soc., Wash., DC (1998).
43. Liu, Y.A. and Stoller, S.D., "Dynamic programming via static incrementalization," pp. 288-305 in *Proc. of ESOP ′99: European Symp. on Programming,* (Amsterdam, The Netherlands, Mar. 22-26, 1999), *Lec. Notes in Comp. Sci.,* Vol. 1576, ed. S.D. Swierstra, Springer-Verlag, New York, NY (1999).
44. Liu, Y.A. and Stoller, S.D., "From recursion to iteration: What are the optimizations?," pp. 73-82 in *Proc. of the 2000 ACM SIGPLAN Workshop on Part. Eval. and Sem.-Based Prog. Manip. (PEPM ′00),* (Boston, MA, Jan. 22-23, 2000), ACM, New York, NY (2000).
45. Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools," in *Proc. of the First Conf. on Empirical Studies of Programming,* (June 1986), Ablex Publishing Co. (1986).
46. McCurdy, A.C., "Accurate computation of divided differences," Ph.D. diss. and Tech. Rep. UCB/ERL M80/28, Univ. of Calif.−Berkeley, Berkeley, CA (1980).
47. Milner, R., "A theory of type polymorphism in programming," *J. Comp. Syst. Sci.* **17** pp. 348-375 (1978).
48. Mogensen, T., "The application of partial evaluation to ray-tracing," Masters thesis, Datalogisk Institut, Univ. of Copenhagen, Copenhagen, Denmark (1986).
49. Ning, J.Q., Engberts, A., and Kozaczynski, W., "Automated support for legacy code understanding," *Commun. ACM* **37**(5) pp. 50-57 (May 1994).
50. Opitz, G., "Steigungsmatrizen," *Zeitschrift fuer angewandte Mathematik und Mechanik* **44** pp. T52-T54 (1964). (In German. English translation available at http://www.cs.wisc.edu/wpis/papers/opitz.zamm64.ps.)
51. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proc. of the ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Softw. Develop. Env.,* (Pittsburgh, PA, Apr. 23-25, 1984), *SIGPLAN Not.* **19**(5) pp. 177-184 (May 1984).
52. Paige, R., *Formal Differentiation – A Program Synthesis Technique,* UMI Research Press, Ann Arbor, MI (1981).
53. Paige, R. and Koenig, S., "Finite differencing of computable expressions," *ACM Trans. Program. Lang. Syst.* **4**(3) pp. 402-454 (July 1982).
54. Paige, R., "Transformational programming—applications to algorithms and systems," pp. 73-87 in *Conf. Rec. of the Tenth ACM Symp. on Princ. of Prog. Lang.,* (Austin, TX, Jan. 24-26, 1983), ACM, New York, NY (1983).

55. Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P., *Numerical Recipes in C: The Art of Scientific Computing,* 2nd. Ed., Cambridge Univ. Press, Cambridge, England (1992).

56. Rall, L.B., *Automatic Differentiation: Techniques and Applications*, *Lec. Notes in Comp. Sci.,* Vol. 120, Springer-Verlag, New York, NY (1981).

57. Rall, L.B., “Differentiation and generation of Taylor coefficients in Pascal-SC,” pp. 291-309 in *A New Approach to Scientific Computation*, ed. U.W. Kulisch and W.L. Miranker, Academic Press, New York, NY (1983).

58. Rall, L.B., “Differentiation in Pascal-SC: Type GRADIENT,” *ACM Trans. Math. Softw.* **10** pp. 161-184 (1984).

59. Rall, L.B., “The arithmetic of differentiation,” *Mathematics Magazine* **59** pp. 275-282 (Dec. 1986).

60. Rall, L.B., “Differentiation arithmetics,” pp. 73-90 in *Computer Arithmetic and Self-Validating Numerical Methods*, ed. C. Ullrich, Academic Press, New York, NY (1990).

61. Rall, L.B., “Point and interval differentiation arithmetics,” pp. 17-24 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G.F. Corliss, Soc. for Indust. and Appl. Math., Philadelphia, PA (1992).

62. Rall, L.B. and Reps, T.W., “Algorithmic differencing,” in *SCAN 2000: 9th GAMM-IMACS Int. Symp. on Sci. Comput., Comp. Arith., and Validated Numerics,* (Karlsruhe, Ger., Sept. 19-22, 2000), (2000). (To appear.)

63. Reps, T. and Turnidge, T., “Program specialization via program slicing,” pp. 409-429 in *Proc. of the Dagstuhl Seminar on Partial Evaluation,* (Schloss Dagstuhl, Wadern, Ger., Feb. 12-16, 1996), *Lec. Notes in Comp. Sci.,* Vol. 1110, ed. O. Danvy, R. Glueck, and P. Thiemann, Springer-Verlag, New York, NY (1996).

64. Shamseddine, K. and Berz, M., “Exception handling in derivative computation with nonarchimedean calculus,” pp. 37-51 in *Computational Differentiation: Techniques, Applications, and Tools*, ed. M. Berz, C. Bischof, G.F. Corliss, and A. Griewank, Soc. for Indust. and Appl. Math., Philadelphia, PA (1996).

65. Speelpenning, B., “Compiling fast partial derivatives of functions given by algorithms,” Ph.D. diss., Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL (Jan. 1980).

66. Veldhuizen, T., “Expression templates,” *C++ Report* **7**(5) pp. 26-31 (June 1995).

67. Weiser, M., “Program slicing,” *IEEE Trans. on Softw. Eng.* **SE-10**(4) pp. 352-357 (July 1984).

68. Wengert, R.E., “A simple automatic derivative evaluation program,” *Commun. ACM* **7**(8) pp. 463-464 (1964).

69. Zippel, R., Personal communication to T. Reps. July 1996.