Building Algebra Systems by Overloading Lisp: Automatic Differentiation

Richard Fateman
Computer Science
University of California
Berkeley, CA, USA

September 17, 2006

Abstract

In an earlier paper [4] we began a discussion of the use of overloaded languages for support of computer algebra systems. Here we extend that notion to provide a more detailed approach to Automatic Differentiation or Algorithm Differentiation (AD).

This paper makes three points. 1. It is extremely easy to do express AD by overloading in Common Lisp. 2. While the resulting program is not the most efficient approach in terms of run-time, it is quite small and very general. It also interacts nicely with some other kinds of generic arithmetic. 3. A more efficient AD compile-time program generation approach is described as well.

1 Introduction

Given a programming language intended to manipulate primarily numbers, arrays of numbers, and other data objects such as strings, we can consider extending the language so that a new set of objects, call them derivative-function pairs. We will try to use them in as many contexts as possible, in particular in place of numbers. That is, just as we can routinely add $2 + 2$ to get 4, the extended language should be able to $x + x$ to get $2x$. Tools for such an enhancement makes it easy to build AD programs. If instead of just representing the function x, we also represent its derivative, namely 1, we can add it to other similar objects, and by adding the derivatives, get the derivative of the sum. This simple idea, pushed to various extremes, turns out to be rather useful.

Many of the issues related to generic or overloaded arithmetic have been discussed in an earlier paper [4], and so will not be emphasized here. We instead proceed to discuss differentiation.

2 Symbolic Differentiation and AD

In this section we motivate the "automatic differentiation" (AD) idea to follow. We contrast it with the notion of symbolic differentiation.

The Lisp programming language has a number of strengths, prominent among them the ease with which it can be used for prototyping other programming languages. However, in brief surveys of languages it is often characterized (or even dismissed) as a language which is suitable for computing "symbolic differentiation".

In fact, the compact representation for a symbolic differentiation program is cited as one driving application for the original Lisp language design. In a conventional setup for Lisp, $x \sin x \log x + 3$ would be written

as `(+ (* x (sin x) (log x)) 3)`[1] A brief program[2] can differentiate this with respect to `x` to get

```
(+ (* (* x (sin x) (log x))
      (+ (* 1 (expt x -1)) (* (* (cos x) 1) (expt (sin x) -1))
         (* (* (expt x -1) 1) (expt (log x) -1))))
   0)
```

an answer which is correct but clumsy in appearance, even after a conversion to more conventional infix:
`(x*sin(x)*log(x))*(1*x^(-1)+(cos(x)*1)*sin(x)^(-1)+(x^(-1)*1)*log(x)^(-1))+0`.

A proper CAS would contain a simplification program (probably much longer than the differentiation program!) to take the symbolic result and reduce it to simpler terms, removing multiplications by 1 and additions with 0, perhaps cancelling factors, etc. resulting in something like $\log x \sin x + \sin x + x \cos x \log x$

This differentiation program `d` takes an expression and a symbol, and computes a derivative. To be more precise and pedantic, the program identifies a Lisp symbol say `x` with a corresponding mathematical indeterminate, say x, and the Lisp expression is based on a recipe which encodes $\sin x$ as `(sin x)` etc. In the short course of this program, it does no harm to conflate these concepts. In fact, the Lisp program is pretty good this way: even setting `x` to 3 does not "confuse" it into thinking it should take the derivative with respect to 3, if that had any meaning. Lisp does not mistake a symbol for its value.

3   Introduction to Automatic Algorithm Differentiation (AD)

Automatic Differentiation or AD has an extensive literature of its own, mostly quite separate from Lisp, and AD provides features that are quite different from the previously described symbolic differentiation[3]. The mostly-true one-sentence summary: AD will read the text of a FORTRAN program $f(x)$ that computes some result k and will write the text of a new program $g(x)$ that computes a pair: k and dk/dx.

But, you may say, that's not really possible, is it? Well, to the extent possible, AD tries to do it. Philosophically, all non-constant functions on floating-point objects represent discrete (not continuous) mappings, and so they have no derivatives. This detail is only one of several "in principle" problems usefully ignored "in practice."

For a complete introduction to the topic of Automatic Differentiation as used in this paper we recommend a visit to the website `www.autodiff.org`. Here you can find a collection of links to the AD standard literature, including AD programs and applications. There are also links to recent conference publications and articles.

There are two major variants of AD techniques, "forward" and "reverse" differentiation. Although we have programmed both, in this paper we will discuss only forward, because it fits more nicely into this overloading concept. For brevity we have name our system ADIL for AD in Lisp.

3.1   A brief defense of Forward Differentiation for ADIL

Consider a space of "differentiable functions evaluated at a point c." In this space we can represent a "function f at a point c" by a pair $\langle f(c),\ f'(c)\rangle$. That is, in this system every object is a pair: a value $f(c)$ and the derivative with respect to its argument $D_x f(x)$, evaluated at c. (written as $f'(c)$).

For a start, note that every number n is really a special case of its own Constant function, $C_n(x)$ such that $C_n(x) = n$ for all x. $C_3(x)$ is thus $\langle 3,\ 0\rangle$. The constant π is $t_1 = \langle 3.14159265\cdots, 0.0\rangle$, which represents a function that is always π and has zero slope. The object $t_2 = \langle c, 1\rangle$ represents the function $f(x) = x$ evaluated at c.

---

[1] As mentioned earlier, there are parsers from infix available.
[2] see Appendix 1
[3] At least initially

At this point we must be clear that all our functions are functions of the *same variable*, and that furthermore we will be fixing a point $x = c$ of interest. It does not make sense to operate collectively on $\langle f(y), D_y f(y) \rangle$ at $y = a$ and $\langle g(x), D_x g(x) \rangle$ at $x = b$.[4]

For example, sin operating on t_2 is the pair $\langle \sin(c), \cos(c) \rangle$. In general, $\sin(\langle a, a' \rangle)$ is $\langle \sin(a), \cos(a) \times a' \rangle$.

We can compute other operations unsurprisingly, as, for example the sum of two pairs: $\langle a, a' \rangle + \langle b, b' \rangle = \langle a+b, a'+b' \rangle$. *Note, we have abused notation somewhat: The "+" on the left is adding in our pair-space, the "+" on the right is adding real numbers. Such distinctions are important when you write programs!* Similarly, the product of two pairs in this space is $\langle a, a' \rangle \times \langle b, b' \rangle = \langle a \times b, a \times b' + a' \times b \rangle$. This can be extended to many standard arithmetic operations in programming languages, at least the differentiable ones[7]. AD implementors seek to find some useful analogy for other operations which do not have obvious derivatives. Falling in this category are most data structure manipulations, calls to external routines, loops, creating arrays, etc.[5] In ADIL, these mostly come free.

3.2   AD as Taylor series

AD technology is not "finding symbolic derivatives" but from a CAS perspective is closer to performing arithmetic with truncated Taylor series. AD converts a *conventional program* computing with conventional scalar values like u to a *new AD program* operating pairs such as $p = \langle u, v \rangle$ representing that scalar function $u(t)$ and its derivative $v(t)$ with respect to an implicit parameter, say t at some (presumably numeric) point $t = c$ In particular, a constant k looks like $\langle k, 0 \rangle$, and the parameter t at the point c looks like $\langle c, 1 \rangle$. That is, $dt/dt = 1$ As a Taylor series, the pair p represents $s = u + v\,t + \cdots$. The equivalence to computation with Taylor series is perhaps the clearest guidance as to how to compute with these pairs. (Generally a CAS Taylor series program can provide an appropriate result, at least for functions with continuous derivatives.) The Taylor expansion of $\cos s$ is $\cos u - \sin u \, v\,t + \cdots$ and so $\cos p = \langle \cos u, -v \times \sin u \rangle$. Some programming operations have discontinuities; if their derivatives are needed then some escape must be arranged: perhaps an error is signalled, or more often a derivative is defined by some pragmatic justification.

The extension of this AD idea to produce programs for higher derivatives is straightforward although perhaps tedious. Here is how it could work: if we need the nth derivatives, we use $n + 1$-tuples. E.g. if the second derivative is called w: If $p = \langle u, v, w \rangle$, compute the Taylor expansion of each operator. Then to continue this example, $\cos p$ yields:

$$\cos p = \cos u - \sin u \, v\,t - \frac{\left(\cos u \, v^2 + 2 \sin u \, w \right) t^2}{2} + \cdots$$

The result triple consists of the coefficients of different powers of t. For a given set of values for u, the program computing those coefficients naturally will need to compute $\sin u$ and $\cos u$ only once. An orthogonal idea, the extension of AD for expressions that rely on m variables is also straightforward; using both ideas together requires a matrix of $(n + 1) \times m$ entries.

3.3   ADIL limitations

Let us be up-front about several limitations.

1. If you have a scientific program written in C or C++ or most likely, FORTRAN, you will not be initially attracted to a tool that works on Lisp programs. (There are translators from FORTRAN to Lisp (f2cl) that have been used for automatic translation of large libraries: so if there is an additional rationale to use Lisp for part of the computation it is not out of the question.)

---

[4]We have not excluded the possible encoding of a vector x, however.

[5]It is a mistake to `declare` variables to be (say) double-floats, when in the ADIL framework they will be `df` structures. We are not aware of any other systematic problems.

2. The AD computer program's running time is slower, but should be no more than a small multiple of the ordinary program. There is a way of trading time for space by using "reverse" AD, but we will not discuss that here. The overloading scheme is slower (we discuss a faster way, soon).

3. Neat things like arithmetic on exact integers or rationals or symbols is not possible in FORTRAN. Supplying the opportunity to use such language features even in cases when they are not needed may make possible certain optimizations of FORTRAN that are difficult in Lisp. If speed is critical, this too may be an issue. Often run-time speed is sometimes thought to be critical even when other issues dominate the time or cost to a solution[6].

On the other hand, just because we don't expect people to shift from FORTRAN to Lisp to use this package does not prevent us from discussing the design, especially since it looks so much nicer than corresponding programs in other languages. ADIL has, among several major advantages, the obvious one of not having to parse its own programs to represent them as data.

3.4   ADIL implementation

There are two fundamentally different approaches to building a forward AD system, and a third that looks plausible, at least briefly, for Lisp.

1. We can take a text for a program P in (more-or-less) unchanged form and by *overloading* each of its operations, make P executable in another domain in which those scalars that represent functions of the distinguished variable of the derivative, are replaced by pairs.

2. We can transform the source code of P into another program in which computations are "doubled" in a particular way to compute the original function as well as the requested derivative.

3. A third technique, usually ignored, could be used in Lisp: write a new version of Lisp's `eval` to do AD. This is not a good idea, even in Lisp, because most Lisp programs don't use `eval`: they are compiled into assembler. We don't want to slow down normal programs doing normal things by running them through *any* version of `eval`. The technique of overloading operators is conceptually similar to patching the `eval` program anyway, and so we will not pursue this option in this paper.

We provide code for both of the first two approaches. Because it inherits all the Lisp facilities not specifically shadowed by AD variations, the overloading method has the advantage of covering just about everything one can do in Lisp. The code is structured in a way that is fairly easy to understand.

Overloading is unfortunately slower in execution compared to the second method, which tends to run at about the minimal extra cost necessary: a (small) multiple of the original code's speed. But source code transformation is ticklish, requiring attention to almost every aspect of the language. Thus our compiler version `dcomp` does not quite cover as much of Lisp as overloading. The two techniques can, fortunately, be used together, so that parts of the code can be transformed for extra speed; these eligible parts are typically the more costly ones anyway: expression evaluation of the composition of built-in functions (or in a specified way, "friendly" functions). That is, certain of the ADIL run-time generated code can be compiled in-line.

For the purpose of overloading, we must decide how to wrap up the pair of function value and derivative. We choose a `defstruct` named `df` with two parts, `f` and `d`, though if we wanted higher derivatives, we might use a vector. A `df` is easily distinguished from a conventional scalar, at runtime, being a different type, and so methods for operations like "+" and "*" can be overloaded. (The source-code transformation technique we describe later does not require these structures; the generated code carries along near-duplicate names, e.g. if the original program has variable `v`, we generate `v_diff_x` (etc.))

---

[6]Benchmarking for speed, even acknowledging that it is subject to abuse, is still more easily quantifiable than say, correctness, generality, or modularity. Thus speed becomes a proxy for all good qualities. And anything that does not improve speed is thereby questionable. Thus we have seen programmers write in "C" because it is faster than some other more convenient language. This would, logically, drive all programmers to write in assembler, but somehow it doesn't.

3.5   Overloading for AD

In this section we describe in more detail how we overload the arithmetic operators and let the rest of the language be inherited from Lisp's standard implementation. (or other variations on generic arithmetic!)

Here's the beginning of that package declaration needed for AD. It could be extended easily. (For example, to add the hyperbolic tangent requires only one line).

```
(defpackage :ga ;generic arithmetic for AD
  (:shadow "+" "-" "/" "*" "expt"      ;binary arith
          "=" "/=" ">" "<" "<=" ">="   ;binary comparisons
          "sin" "cos" "tan"            ;... more trig
          "atan" "asin" "acos"         ;... more inverse trig
          "sinh" "cosh" "atanh"        ;... more hyperbolic
          "expt" "log" "exp" "sqrt"    ;... more exponential, powers
          "1-" "1+" "abs"              ;... odds and ends
          )
  (:use :common-lisp))
```

The implementation of AD looks like any system that overloads a generic arithmetic package, shadowing all the built-in arithmetic with new methods that do the arithmetic on pairs. Comparisons, however, need some work. We defined `two-arg-` variants for `df` pairs by ignoring the derivative and looking only at the function value. To accomplish this we wrote macro programs so that for example, the macro-expansion of (`defcomparison /=`), says all we need to say about `/=`, the not-equal function. We also used macro-expansions to set up programs based on the derivative property for each operator. This looks like a single-argument differentiable function. To define all that our `df` handling needs to know about the sin function, we must say no more than (`r sin (cos x)`). The `r` macro expansion defines the necessary piece of code for the chain rule.

Other than these one-argument functions we need to write some specific programs for `two-arg-+` etc. Our general arithmetic in Lisp allows for any number of arguments for functions where it makes sense; (`+ x y z`) is macro-expanded to two calls to `two-arg-+` before being compiled.

This AD-overload system constitutes about 130 lines of Lisp code other than the package declarations and some examples.

3.6   Using ADIL

Consider the function $f(x) = x \sin x \log x + 3$ written in Lisp as
(`defun f(x)(+ (* x (sin x) (log x)) 3)`).
Using the syntax in [4] we could define it by `$f(x):=x sin x log x + 3`. We can evaluate f at a point x where x is 1.23 by (`f (df 1.23 1.0)`)) which returns

```
<3.2399835288524628d0, 1.2227035>
```

In this system we have defined `df` as a constructor taking two numbers. This pair is displayed in angle-brackets. $p = \langle 1.23, 1.0 \rangle$. Note that the text of the program `f` has not changed at all; applying the function to a different type of argument, and running the program within the `:ga` package is all that is needed. Of course the meaning of a function like "+" within the `:ga` package is different from the standard "+" in spite of the observation that the text is the same. (The function `f` can be invoked from Lisp programs in other packages by calling it by a fully-qualified name like `ga::f`.)

A more interesting program is this one.

5

```
(defun s(x) (if (< (abs x) 1.0d-5) x
             (let ((z (s (* -1/3 x))))
               (-(* 4 (expt z 3))
                 (* 3 z)))))
```

Not at all obviously, this computes an approximation to $\sin x$ by applying an identity recursively. That is, $\sin x$ is a polynomial $4z^3 - 3z$ in $z = \sin(-x/3)$, and that for small enough x, $\sin x = x$. Let us try it out by typing (s (df 1.23 1)). We get

```
<0.942489, 0.33423734>
```

This not only provides a value for $\sin(1.23)$ but also computes 0.33423734, the second part of the pair, which happens to be $\cos(1.23)$. It appears that ADIL somehow knew that s computed sin() and therefore it also computed its derivative, cos(). ADIL just followed directions: It computed something that looked like the derivative of sin only because it ran the *derivative of the program that computed something that looked like sin.* Incidentally, if we used (s (df 1.23d0 1)) the answer is computed in double precision.

The classic recursive program in Lisp is factorial:
```
(defun fact(x) (if (= x 1) 1 (* x (fact (1- x)))))
```
which we modify to
```
(defun fact(x) (if (= x 1) (df 1 0.422784335098d0) (* x (fact (1- x)))))
```
whose peculiar base case is now the value one, with a peculiar derivative. We do this so as to make it correspond to the derivative of the Gamma function[7]. With this form we can provide not only the factorial but its "derivative" (at least at integer points). In these examples we have not modified Lisp syntax at all.

Other parts not mentioned in the Lisp language, namely parts of the language not overloaded, are imported without requiring any attention or comment. Thus to compute and print ten $\sin x$ values, we can use dotimes as in
```
(dotimes (i 10) (print (s (df i 1)))).
```

3.7 Newton Iteration, or, Why is AD useful?

The point we wish to make here is that if we are given a complicated function $F : R \to R$ arranged as an expression, and all the sub-functions "cooperate" properly, we can feed in a pair $\langle c, 1 \rangle$, representing the expression x and its derivative with respect to x, namely 1, each evaluated at $x = c$. We get out pairs $\langle f, f' \rangle = F(\langle c, 1 \rangle)$ where the latter is the pair $f = F(c)$, and $f' = D_x(F(x))|_{x=c}$. and this may be exactly what we want in an application. These are discussed in papers available via www.autodiff.org. Here we take only the simplest and easiest to motivate example, Newton iteration.

For example, to converge to a root in a Newton iteration for $f(z) = 0$ given an initial guess c_0 or $t_0 = \langle c_0, 0 \rangle$, we compute $F(t_i) = \langle f(t_i), f'(t_i) \rangle$. Then the next iteration $t_{i+1} = t_i - f(t_i)/f'(t_i)$. If this is not sufficiently accurate we consider repeating with $\langle f, f' \rangle = F(t_{i+1})$, etc.

The program is shorter than the explanation.

```
 Newton iteration: (ni fun guess) usage: fun is a
;; function of one argument guess is an estimate of solution of
;; fun(x)=0 output: a new guess. (Not a df structure, just a number)

(defun ni (f z) ;one Newton step
  (let* ((pt (if (df-p z) z (df z 1)))  ; make sure init point is a df
         (v (funcall f pt))) ;compute f, f' at pt
    (df-f (- pt (/ (df-f v)(df-d v)))))))
```

---

[7]a continuous version of factorial well-known among the "special functions".

As a simple example, consider $f(x) = sin(1 + 2x)$ or
(defun f(x)(sin(+ 1 (* 2 x)))). then

```
(setf h 2.0d0);; just a guess
(setf h (ni 'f h))
(setf h (ni 'f h))
;; h converges to 1.0707963267948966d0
```

We can write a version of Newton iteration to return the value too. Then both the residual and the derivative can be taken into account in testing whether the Newton iteration has converged sufficiently. That program would look like:

```
(defun ni2 (f z)
  (let* ((pt (if (df-p z) z (df z 1))) ;if z is not a df, make it one
         (v (funcall f pt))) ;compute f, f' at pt
  (values  (df-f (- pt (/ (df-f v)(df-d v)))) ;the next guess
           v))) ; the residual and derivative
```

Now that we have a program for only one step, we can show a program that can use it to find the zero. This is a ticklish proposition because sometimes this iteration does not converge. We have to use some stopping heuristic. We could stop when two successive iterations are close in terms of relative or absolute error, or use some other measure.

A particularly simple iteration driver program uses `ni2` which returns the value of the residual which we test. We also quit with an error message if some count is exceeded.

```
(defun run-newt2(f guess &key (abstol 1.0d-8) (count 18)) ;; Solve f=0
  ;; It looks only at the residual.
    (dotimes (i count  ;; do at most count times. failure prints msg
              (error "~%Newton quits after ~s iterations: ~s" count guess))
      (multiple-value-bind
          (newguess v)
          (ni2 f guess)
        (if (< (abs (df-f v)) abstol)
          (return newguess)
          (setf guess newguess)))))
```

In this Newton iteration example, our program must, by its nature, separately get a value and derivative. We have used three functions particular to ADIL, namely `df-p` a predicate which will return true if applied to a `df` object, as well as the two selection functions, `df-f` and `df-d` for extracting the value and derivative respectively from a `df` object.

3.8   What about speed?

The generic arithmetic (ga) system for ADIL produces code that is slower than ordinary Lisp code, especially if we make efforts to optimize the ordinary Lisp[8].

How much of a speed difference is there? On a variety of benchmarks involving mostly computations of how to dispatch-to-the-right-method, using the generic arithmetic for ADIL seems to be about a factor of 10 over "normal Lisp" and perhaps a factor of 50 over "optimized" Lisp (when that Lisp is constrained say,

---

[8]Note that without type declarations and compilation, ordinary Lisp already has its own burden of generic arithmetic. Short and long (arbitrary length) integers, single or double floats, rationals, and complex numbers, as well as all plausible combinations are handled by standard ANSI Common Lisp. With appropriate instructions to the compiler Common Lisp arithmetic can be compiled to "non-generic" straight-line floating-point code, comparable to that of other high-level languages.

to double-floats.) On tests where most of the work is done in subroutines such as sin or log, the difference is much less: the log routine takes the same time whether it is called from the `ga` package or from the `user` package. On parts of tests where the operations are *other than generic arithmetic* such as looping over indexes, the ADIL or `ga` programs run at full speed because they are in fact running identical instructions.

3.9   Source code transformation for AD

As mentioned earlier, we wrote another part of ADIL, `dcomp` that can compile programs, in-line, in a restricted language subset of Lisp, essentially that of functional-style arithmetic programs. In this situation, our benchmarking a simple example suggests that the comparison between the ordinary Lisp for computing a function f and the ADIL Lisp for computing a function f and its derivative is a small factor; we have observed a typical factor of about two; we expect perhaps a factor of up to 5, judging from the literature.

   As an experiment we compute $3 + z*(4 + z)$ where $z = \sin x$. The `dcomp` version is shown with the `defdiff` defining form. All code is run through a Lisp compiler before timing, and the reported times are for a run of 10,000 computations. The first set shows that the code transformation provides function and derivative values about 2.3 times slower than the fastest program we could reasonably expect to provide just the function value. If we were computing both the function and its derivative with separate programs, we would expect a factor of 2.

```
(defun kk(x &aux z)   ;;optimized version
  (declare (double-float x z)(optimize (speed 3)(safety 0)))
  (setf z (sin x))
  (+ 3.0d0 (* z (+ 4.0d0 z))))

(defdiff kz(x)
  (progn (setf z (sin x))
         (+ 3.0d0 (* z (+ 4.0d0 z)))))

(kk 1.2d0) -->    30 ms  /no derivs!
(kz 1.2d0) -->    70 ms  /with derivs!!
```

In the next example we use the same function, but define it in the generic arithmetic package. By calling it on different types we get different behavior.

```
(defun kk(x &aux z) (setf z (sin x))(+ 3.0d0 (* z (+ 4.0d0 z))))

(kk (   1.2d0)) -->  1523 ms /no derivs!  ;  51 times slower
(kk (df 1.2d0)) --> 40100 ms /with derivs ; 573 times slower
```

The time spent in the overloaded model is substantially higher. Fortunately the program `kz` could be used instead of `kk` as a subroutine if one is interested in running a presumably much more elaborate computation using the overloading methods.

   The body of `kz` which is produced can be examined as transformed Lisp before it is compiled. This can be done by tracing a program `dc` which is internal to the `dcomp` ADIL. This shows:

```
  (lambda (g94)
        "(progn (setf z (sin x)) (+ 3.0d0 (* z (+ 4.0d0 z)))) wrt x"
        (declare (double-float g94))
```

8

```
(declare (optimize (speed 3) (debug 0) (safety 0)))
(let ((t102 0.0d0) (f101 0.0d0)
      (t100 0.0d0) (f99 0.0d0)
      (t98 0.0d0)  (f97 0.0d0)
      (z_DIF_x 0.0d0) (t96 0.0d0)
      (f95 0.0d0))
  (declare (double-float t102 f101 t100 f99 t98 f97 z_DIF_x t96 f95))
  (setf f95 (sin g94))
  (setf t96 (cos g94))
  (setf z f95)
  (setf z_DIF_x t96)
  (setf t98 0.0d0 f97 3.0d0)
  (setf t100 z_DIF_x f99 z)
  (setf t102 0.0d0 f101 4.0d0)
  (setf t102 (+ z_DIF_x t102))
  (setf f101 (+ z f101))
  (setf t100 (+ (* f101 t100) (* t102 f99)))
  (setf f99 (* f101 f99))
  (setf t98 (+ t100 t98))
  (setf f97 (+ f99 f97))
  (df f97 t98)))
```

What else can `dcomp` do? In addition to the usual arithmetic operations and built-in functions (sin, cos, log, etc.) as declared in the generic arithmetic package, `dcomp` can handle `if, progn, setf`. However, `dcomp` is not as general as overloading, at least not as we have implemented it. Functions defined with `defdiff` have one distinguished argument, the first, that is the point-value of x (the derivative variable) with an implicit derivative of 1. That value is a numeric argument. There may be additional arguments of any kind. Functions defined with `defdiff` return only `df` structures. Because the derivative is always with respect to the first formal argument, `(defdiff f(x y z)(cos (+ x y z)))` is legal, but only one derivative (with respect to x) is computed. This is not an essential restriction, but would require a more length specification (on how to pass additional arguments and additional derivatives, and somewhat more programming to implement.

While `dcomp` could be expanded to cover the same ground as the overloaded option, we have chosen a compromise of less functionality for run-time speed and simplicity of implementation. Within the `dcomp` framework it is difficult to implement full recursion, basically because our programs must understand the `df` output forms, so only a restricted version (essentially so-called tail-recursion) is possible. We found that specifying the restriction on recursion (or alternatively, removing the restriction!) led to too much complexity and so we opted to simply forbid it. Recursive functions can easily use the overloaded ADIL methods.

How does this fit into the generic framework, then? As shown in the timings, a function like `f` above, can run at full speed.

The `dcomp` file is about 290 lines of Lisp code. It would not be difficult to alter the code-generation part of dcomp to produce FORTRAN or C function subroutines, and so one could argue that we could do this task, starting and ending with FORTRAN, by transmuting the middle of the AD processing into Lisp; we expect that a the middle part of the code used in other AD tools is, in effect, simulating what we do in Lisp.

4 Does it really make sense to use ADIL?

For fans of Lisp, there is no question that one motivation is to show off the natural advantage of the language: Lisp provides a natural representation for programs as data and a natural form for writing programs that

write programs, which is what we do in ADIL. It also has an unobtrusive object-oriented programming system, only part of which is used here. We used subsetting of types and dispatched on methods based on the types of *all* the parameters, not just the methods associated with the type of the "first argument".

The code is short, and is in ANSI standard Common Lisp. It should run without any change in any conforming system. While it is not using the most obvious idioms of introductory Lisp, its more sophisticated style results in more compact code. There are now several excellent books on Common Lisp; some people have argued that reading them will make you a better programmer in any language.

For persons only slightly familiar with Lisp, or whose acquaintance comes from one of the too-common texts that use the Lisp 1.5 of 1959 as the definition, a glance at the Lisp shows how short such code can be, and might serve as an encouragement to learn more about modern Common Lisp. If you are familiar with the complexities of automatic differentiation when presented in another language, the relatively brevity should also be observable.

If you care not a whit about Lisp or implementation strategies, you may prefer to refer to the recently-revised online documentation for ADIFOR 2.0 version D, some 99 pages, and read in detail how programs to be differentiated must be distinguished from ordinary FORTRAN (77) programs. Using ADIFOR requires declaring and marking program variables, adjusting numerous parameters, perhaps revising the FORTRAN in order to obey various restrictions, and following detailed guidelines on the use of these programs. While providing the same level of coverage of attention to detail would undoubtedly increase the size of ADIL, we expect the code size to remain rather small.

The flexibility one gets is apparent by looking at the code in which Lisp macro-definitions have made the addition of new derivative information simple. For example, to insert a new rule for differentiation of $\tanh x$ one adds `"tanh"` to the `shadow` list, and execute
`(r tan (expt (cosh x) -2))`.
By comparison, large and monolithic systems are relatively rigid, and require some effort to port, extend or optimize: the original designers must anticipate the spectrum of possible choices, perhaps freezing some choices, and then describe all the variants in detail. Each change in the system typically requires recompilation and linking of a collection of programs. On the application side, the user then read the details, and hope there are no bugs; the user is unlikely to be able, in any case, to repair them. Some of the restrictions of the large systems seem to be arbitrary–perhaps a small point, but it did not occur to us to have to exclude recursive functions, although recursion is a feature lacking in ADIFOR.

Finally, we wish to point out, as we have before, that Lisp is perfectly adequate for expressing numeric computation. While it is possible to find a slow Lisp interpreter, almost all systems provide compilers, and some are quite sophisticated in generating efficient code.

5   Composing Overloads

In our earlier paper we illustrated overloading of *several types simultaneously*. We repeat them here for coherence, but with the emphasis on df.

In these examples we show that combining special operations *can be* as simple as composing them. Here is a recursive definition of Legendre polynomials of the first kind.

```
(defun p(m x)(cond ((= m 0) 1)
                   ((= m 1) x)
                   (t (*  (/ 1 m) (+ (* (1- (* 2 m)) x (p (1- m) x))
                                     (* (- 1 m) (p (- m 2) x)))))))
```

Here are some possible ways of using it in the generic arithmetic package.

```
: (p 3 1/2)               ;exact value of legendre[3,1/2]
-7/16
```

```
: (p 3 0.5)              ;single-float value
-0.4375
: (p 3 (df 0.5 1.0))     ;single-float value, as well as derivative
<-0.4375, 0.375>
: (p 3 (df 0.5d0 1.0))   ;double-float value and derivative
<-0.4375d0, 0.375d0>
: (p 3 (df 1/2 1))       ;exact value and derivative
<-7/16, 3/8>
: (p 3 \%x)              ;exact value in terms of x
(1/3)*((5*x)*((1/2)*((-1)+(3*x)*x))+(-2)*x)
: (simp (p 3 \%x))       ;exact value, somewhat simplified
(1/3)*((5/2)*x*((-1)+3*x^2)+(-2)*x)
: (simp (p 3 (df \%x 1)));exact value and derivative, simplified
<(1/3)*((5/2)*x*((-1)+3*x^2)+(-2)*x),
 (1/3)*((-2)+15*x^2+(5/2)*((-1)+3*x^2))>
: (ratexpand (p 3 (df \%x 1)))
<(1/2)*((-3)*x+5*x^3), (1/2)*((-3)+15*x^2)>
                         ;same answer, using canonical rational expansion
```

There are a number of kinds of arithmetic overloads mentioned in our earlier paper; this paper has discussed in detail one of the more unusual ones.

6   Conclusion

Generic arithmetic can be easily supported in Common Lisp. We show how it can support Automatic Differentiation (AD). These results are not surprising, though we hope that our presentation has some novelties, in particular the brevity and effectiveness of the program. What is perhaps surprising is that no one has written this paper earlier, since the Lisp language appears to be such a fine host for this effort.

Appendix 1

First we display a seven line Lisp differentiation program (similar to many others written over the years) that is distinguished primarily by brevity. This one was posted on a Lisp newsgroup by Pisin Bootvong, recently, and makes use of the Common Lisp object system (CLOS) and `destructuring-bind` nicely:

```
(defmethod d ((x symbol) var) (if (eql x var) 1 0))
(defmethod d ((x number) var) 0)
(defmethod d ((expr list) var)
   (destructuring-bind (op e1 e2) expr
     (case op
       (+ '(+ ,(d e1 var) ,(d e2 var)))
       (* '(+ (* ,(d e1 var) ,e2) (* ,e1 ,(d e2 var)))))))
```

Here's a more elaborate, but still short Lisp program with more capabilities and greater extensibility [2].

```
(defun d(e v)(if(atom e)(if(eq e v)1 0)
              (funcall(get(car e)'d #'(lambda (e v) '(d ,e, v))) e v)))

(defmacro r(op s)'(setf(get ',op 'd) ;;define a rule to diff operator op!
```

11

```
                (compile() '(lambda(e v)
                           (let((x(cadr e)))
                             (list '* (subst x 'x ',s) (d x v)))))))))
(r cos (* -1 (sin x)))
(r sin (cos x))
(r exp (exp x))
(r log (expt x -1)) ;; etc,
(setf(get '+ 'd)  ;; rules for +, *, handle n args, not just 1
  #'(lambda(e v) '(+,@(mapcar #'(lambda(r)(d r v))(cdr e)))))
(setf(get '* 'd)
  #'(lambda(e v) '(*,e(+,@(mapcar #'(lambda(r) '(*,(d r v)(expt,r -1)))(cdr e))))))
(setf(get 'expt 'd)
  #'(lambda(e v) '(*,e,(d '(*,(caddr e)(log,(cadr e)))v))))
```

Other programming languages, especially ones with a "functional" approach, can usually handle this task nicely, but the major issue (and one not addressed by any of the programs illustrated here) is simplification of the result. Trying to write a simplifier adds substantially to the programmer's burden. The differentiation program in a computer algebra system like Macsyma is much larger, not only because it handles a larger class of functions, but also because it trades code-size for speed. It avoids generating such naive forms intersperses simplifying along the way.

Another program whose specifications seem superficially like the previous one does not build any list structure. It assume that the result of interest is the value of the derivative at a point, not its symbolic representation. Thus the d function takes another argument, the point p. It returns the derivative of the expr with respect to the var at the point p. We are no longer returning lists. Note that we now have to evaluate expressions involving the variable, for which we have defined the val method. Although such a program can be written entirely using the skeleton of the previous few programs, we illustrate a different approach using generic programming (a handle on object-oriented programs) supported in Common Lisp.

```
(defmethod d ((x symbol) var p) (if (eql x var) 1 0))
(defmethod d ((x number) var p) 0)
(defmethod d ((expr list) var p)
   (destructuring-bind (op e1 e2) expr
     (case op
       (+ (+ (d e1 var p) (d e2 var p)))
       (* (+ (* (d e1 var p) (val e2 var p)) (* (val e1 var p) (d e2 var p)))))))

(defun val(expr var p)(funcall  '(lambda (,var) ,expr) p))
```

This program's case statement would have to be expanded for other two-argument functions, and would also need to be altered for one-argument functions like sin and cos.

Appendix 3 DF

```
;;; Automatic Differentiation code for Common Lisp (ADIL)
;;; using overloading, forward differentiation.

;; code extended by Richard Fateman, November, 2005

(defpackage :df ;derivative and function package
  (:use  :cl)
```

```
   (:shadowing-import-from
    :ga
    "+" "-" "/" "*" "expt" ;binary arith
    "=" "/=" ">" "<" "<=" ">=" ;binary comparisons
    "sin" "cos" "tan" ;... more trig
    "atan" "asin" "acos" ;... more inverse trig
    "sinh" "cosh" "atanh" ;... more hyperbolic
    "expt" "log" "exp" "sqrt" ;... more exponential, powers
    "1-" "1+" "abs" "incf" "decf"
    "numerator" "denominator"
    "tocl"    )
   (:export "df")
   )

(require "ga" ) ;generic arithmetic framework
(provide "df" )
(in-package :df)


;;  structure for f,d: f is function value, and d derivative, default 0
;; df is also the constructor for an object with 2 components, f and d..
(defstruct (df (:constructor df (f &optional (d 0)))) f d )
(defmethod print-object ((a df) stream)(format stream "<~a, ~a>" (df-f a)(df-d a)))

;;comparison of df objects depends only on their f parts (values)
;;extend the generic arithmetic for this purpose

(defmacro defcomparison (op)
  (let ((two-arg (intern (concatenate 'string "two-arg-"
     (symbol-name op))    :ga ))
        (cl-op (tocl op)))
    `(progn
       ;; only extra methods not in ga are defined here.
       (defmethod ,two-arg ((arg1 df) (arg2 df))    (,cl-op (df-f arg1)(df-f arg2)))
       (defmethod ,two-arg ((arg1 number) (arg2 df))(,cl-op arg1(df-f arg2)))
       (defmethod ,two-arg ((arg1 df) (arg2 number))(,cl-op (df-f arg1) arg2 ))
       (compile ',two-arg)
       (compile ',op)
       ',op)))

(defcomparison >) ;; define two-arg-> for df and other types
(defcomparison =)
(defcomparison /=)
(defcomparison <)
(defcomparison <=)
(defcomparison >=)

;; extra + methods specific to df
(defmethod ga::two-arg-+ ((a df) (b df))    (df  (+ (df-f a)(df-f b))
 (+ (df-d a)(df-d b))))
```

```
(defmethod ga::two-arg-+ ((b df)(a number))   (df  (+ a (df-f b))    (df-d b)))
(defmethod ga::two-arg-+ ((a number)(b df))   (df  (+ a (df-f b))    (df-d b)))


;;extra - methods

(defmethod ga::two-arg-- ((a df) (b df))    (df  (- (df-f a)(df-f b))
 (- (df-d a)(df-d b))))
(defmethod ga::two-arg-- ((b df)(a number))   (df  (-  (df-f b) a)    (df-d b)))
(defmethod ga::two-arg-- ((a number)(b df))   (df  (- a (df-f b))    (df-d (- b))))


;;extra * methods
(defmethod ga::two-arg-* ((a df) (b df))
  (df  (* (df-f a)(df-f b))
       (+ (* (df-d a) (df-f b)) (* (df-d b) (df-f a)))))
(defmethod ga::two-arg-*
    ( (b df)(a number)) (df  (* a (df-f b))  (* a (df-d b))))
(defmethod ga::two-arg-*
    ((a number) (b df)) (df  (* a (df-f b))  (* a (df-d b))))


;; extra divide methods
(defmethod ga::two-arg-/  ((u df) (v df))
  (df  (/ (df-f u)(df-f v))
    (/ (+ (* -1 (df-f u)(df-d v))
  (* (df-f v)(df-d u)))
    (* (df-f v)(df-f v)))))
(defmethod ga::two-arg-/  ((u number) (v df))
  (df  (/ u (df-f v))
    (/ (* -1  (df-f u)(df-d v))
    (* (df-f v)(df-f v)))))
(defmethod ga::two-arg-/  ((u df) (v number))
  (df  (/ (df-f u) v)
    (/ (df-d u) v)))


;; extra expt methods
(defmethod ga::two-arg-expt  ((u df) (v number))
  (df  (expt (df-f u) v)
       (* v (expt (df-f u) (1- v)) (df-d u))))

(defmethod ga::two-arg-expt ((u df) (v df))
  (let* ((z (expt (df-f u) (df-f v)));;z=u^v
 (w;;   u(x)^v(x)*(dv*LOG(u(x))+du*v(x)/u(x)) = z*(dv*LOG(u(x))+du*v(x)/u(x))
  (* z (+
(* (log (df-f u)) ;log(u)
   (df-d v)) ;dv
(/ (* (df-f v)(df-d u)) ;v*du/ u
   (df-f u))))))
    (df  z  w)))

(defmethod ga::two-arg-expt ((u number) (v df))
```

```
  (let* ((z (expt u (df-f v)))) ;;z=u^v
 (w   ;;     z*(dv*LOG(u(x))
  (* z (* (log u) ;log(u)
 (df-d v)))))
    (df  z  w)))

;; A macro definition rule to define rules.

(defmacro r (op s)
  `(progn
     (defmethod ,op ((a df)) ;; the chain rule d(f(u(x)))=df/du*du/dx
       (df  (,op (df-f a))
    (* (df-d a) ,(subst '(df-f a) 'x s))))
     (defmethod ,op ((a number)) (,(tocl op) a))))

;; add as many rules as you can think of here.
;; should insert them in the shadow list too.
(r sin (cos x))
(r cos (* -1 (sin x)))
(r asin (expt (+ 1 (* -1 (expt x 2))) -1/2))
(r acos (* -1 (expt (+ 1 (* -1 (expt x 2))) -1/2)))
(r atan (expt (+ 1 (expt x 2)) -1))
(r sinh (cosh x))
(r cosh (sinh x))
(r atanh (expt (1+ (* -1 (expt x 2))) -1))
(r log (expt x -1))
(r exp (exp x))
(r sqrt (* 1/2 (expt x -1/2)))
(r 1-  1)
(r 1+  1)
(r abs x);; hm.

;; some examples of functions that can be differentiated, including recursive factorial
;; A factorial with "right" derivative at x=1 to match gamma function

 (defun fact(x) (if (= x 1) (df 1 0.422784335098d0) (* x (fact (1- x)))))

;; stirling approximation to factorial
(defun stir(n) (* (expt n n) (exp (- n))(sqrt (*(+ (* 2 n) 1/3) 3.141592653589793d0))))

(defun ex(x)(ex1 x 1 15))
(defun ex1(x i lim)  ;; generate Taylor summation exactly for exp()
  (if (= i lim) 0 (+ 1 (* x  (ex1 x (+ i 1) lim)(expt i -1)))))
```

Appendix 3: Some examples and timing

Some timing results suggest that we lose about a factor of 10 in speed by running in a generic arithmetic system. That is, even if we are using (say) floating point numbers. This is the cost of the extra checking,

just in case. Under these conditions, if we actually use `df` numbers, it is perhaps a factor of two additionally slower.

That means the big slowdown is in generic method dispatch for simple methods like addition or multiplication. This is hardly surprising.

If we compile with declarations for fixnum types among the lisp built-ins, we improve by another 30 percent.

```
#|

;; slowmul is a program that multiplies by doing repeated adds.
;; slowmulx is the same program with optimization turned on

(defun slowmul(x y ans)(if (= x 0) ans (slowmul (1- x) y (+ y ans))))

(defun slowmulx(x y ans)
  (declare (fixnum x y ans) ;; or (double-float x y ans)
   (optimize (speed 3)(safety 0)(debug 0)))
   (if (= x 0) ans (slowmulx (1- x) y (+ y ans))))

(time (dotimes (i 10)(slowmul  1000000  2  0))) ;compiled 19.38s in :ga
(time (dotimes (i 10)(slowmul  1000000  2  (df 0)))) ;    24.45s in :ga
(time (dotimes (i 10)(slowmul  1000000  2  0))) ;compiled  .37s in :user
(time (dotimes (i 10)(slowmulx 10000.0d0 2.0d0 0.0d0)));    .38s in :user declared doubles
(time (dotimes (i 10)(slowmulx 1000000 2 0)))         ;      .29s in :user declared fixnums

;; this is an interesting function that numerically computes (sin x)

(defun s(x) (if (< (abs x) 1.0d-5) x
     (let ((z (s (* -1/3 x))))
(-(* 4 (expt z 3))
  (* 3 z)))))

 (s  (df 1.23d0 1.0))   computes both sin and cos of 1.23.

;; runtimes vary as to whether the program is compiled in generic arithmetic (ga)
;; or the ordinary arithmetic (user) package.

(time (dotimes (i 100) (s (df 100000.0d0 1))))   :ga 40ms
(time (dotimes (i 100) (s  100000.0d0  )))        :ga 20ms
(time (dotimes (i 100) (s  100000.0d0  ))) :user 10ms av over more runs
(time (dotimes (i 100) (ss 100000.0d0  ))) :user  8ms av over more runs

;; ss is same as s, but compiled to run faster and to work only on double-floats.

(defun ss(x) (declare (double-float x)
     (optimize (speed 3)(safety 0) (debug 0)))
      (if (< (abs x) 1.0d-5) x (let ((z (ss (* #.(/ -1 3.0d0) x))))
   (-(* 4.0d0 (expt z 3))
     (* 3.0d0 z)))))
```

16

```
;; for additional on-line comments, tests, etc. see
;; www.cs.berkeley.edu/~fateman/generic/df.lisp
```

Appendix 4 Dcomp

```
;; code for dcomp.lisp is about 291 lines, available on request.
;; a more serious version generating code for backwards diff
;; version of AD is also available.
```

References

[1] H. Abelson, G. Sussman, *Structure and Interpretation of Computer Programs,* 2nd edition 1996. MIT Press. Full text available on-line. http://mitpress.mit.edu/sicp/full-text/sicp/book/book.html

[2] "A Short Note on Short Differentiation Programs in Lisp, and a Comment on Logarithmic Differentiation," *SIGSAM Bulletin Volume 32, Number 3*, Sept., 1998,pp. 2-7. `www.cs.berkeley.edu/~fateman/papers/deriv.pdf`.

[3] , R. Fateman and Tak Yan, "Computation with the Extended Rational Numbers and an Application to Interval Arithmetic" 1994// `http://www.cs.berkeley.edu/~fateman/papers/extrat.pdf`.

[4] R. Fateman, "Building Algebra Systems by Overloading Lisp", (submitted for publication)

[5] R. Fateman, "Backward Automatic Differentiation in Lisp," (in progress)

[6] A. Griewank, "On Automatic Differentiation," in M. Iri & K. Tanabe (Eds.) *MATHEMATICAL PROGRAMMING,* Kluwer Academic Publishers, 1989, pp. 83-107.

[7] A. Griewank and G. Corliss (eds), *Automatic Differentiation of Algorithms*, SIAM, 1991. esp. paper by L. Rall

[8] A. Perlis, Itturiaga, R., Standish, T. "A Definition of Formula Algol," in *Proc. Symposium on Symbolic and Algebraic Manipulation of the ACM*, Washington, D.C., March, 1966.

[9] Richard D. Jenks and Robert S. Sutor, AXIOM: *The Scientific Computation System,* Springer-Verlag, 1992.

[10] P. Norvig, *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann, 1992.

[11] R. G. Tobey, "Experience with FORMAC algorithm design", Comm. ACM 9 no. 8 (1966) p. 589–597.

[12] W. T. Wyatt, Jr. D.W. Lozier, and D.J. Orser, "A Portable Extended Precision Package and Library with Fortran Precompiler" ACM Trans on Math Software ISSN:0098-3500 1976, vol. 2 no. 3 p. 209–231.)