1-5-2008

Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1)

Barak A. Pearlmutter
barak@cs.nuim.ie

Jeffrey M. Siskind
Purdue University, qobi@purdue.edu

Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast

(CVS: 1.1)

Barak A. Pearlmutter[1] and Jeffrey Mark Siskind[2]

[1] Hamilton Institute, National University of Ireland Maynooth, Co. Kildare, Ireland
barak@cs.nuim.ie
[2] School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette IN 47907-2035 USA qobi@purdue.edu

Summary. This paper discusses a new AD system that correctly and automatically accepts nested and dynamic use of the AD operators, without any manual intervention. The system is based on a new formulation of AD as highly generalized first-class citizens in a λ-calculus, which is briefly described. Because the λ-calculus is the basis for modern programming-language implementation techniques, integration of AD into the λ-calculus allows AD to be integrated into an aggressive compiler. We exhibit a research compiler which does this integration, and uses some novel analysis techniques to accept code involving free dynamic use of nested AD operators, yet performs as well as or better than the most aggressive existing AD systems.

Key words: Nesting, multiple transformation, forward mode, optimization

1 Introduction

Over sixty year ago, Church [2] described a model of computation which included higher-order functions as first-class entities. This *λ-calculus*, as originally formulated, did not allow AD operators to be defined, but Church did use the derivative operator as an example of a higher-order function with which readers would be familiar. Although the λ-calculus was originally intended as a model of computation, it has found concrete application in programming languages *via* two related routes. The first route came from the realization that extremely sophisticated computations could be expressed crisply and succinctly in the λ-calculus. This led to the development of programming languages (LISP, SCHEME, ML, HASKELL) that themselves embody the central aspects of the λ-calculus, in particular the ability to freely create and apply functions including higher-order functions. The second route arose from the recognition that various program transformations and programming-language theoretic constructs were naturally expressed using the λ-calculus. This resulted in the use of the λ-calculus as the central mathematical scaffolding of programming-language theory (PLT): both as the formalism in which the semantics of programming-language constructs are mathematically defined, and as the intermediate format into which computer programs are converted for analysis and optimization.

A substantial subgroup of the PLT community is interested in advanced or functional programming languages, and has spent the decades since the conception of the LISP programming language [7, 8] and its descendents inventing techniques by which programming languages with higher-order functions can be made efficient. These techniques are part of the body of knowledge we refer to as PLT, and are the basis of the implementation of modern programming-language systems such as JAVA, C♯, the GHC HASKELL compiler, GCC 4.x, etc. Some of these techniques are being gradually rediscovered by the AD community. For instance, a major feature in the TAPENADE AD system [3] is the utilization of a technique by which values to which a newly-created function refer are separated from the code body of the function; this method is used ubiquitously in PLT, where it is referred to as *lambda lifting* or *closure conversion* [5].

We point out that—like it or not—the AD transforms are higher-order functions: functions that both take and return other functions. As such, attempts to build implementations of AD which are efficient and correct encounter the same technical problems which have already been faced by the PLT community. In fact, the technical problems faced in AD are a superset of these, as the machinery of PLT, as it stands, is unable to fully express the reverse AD transformation. We have embarked upon a sustained project to bring the tools and techniques of PLT to bear on AD. To this end, we have found a way to incorporate first-class AD operators (functions that perform forward- and reverse-mode AD) into the λ-calculus. This solves a host of problems: (1) the AD transforms are specified formally and generally; (2) nesting of the AD operators, and inter-operation with other facilities like memory allocation, is assured; (3) it becomes straightforward to integrate these into aggressive compilers, so that AD can operate in concert with optimization rather than beforehand; (4) sophisticated techniques can migrate various computations from run time to compile time; (5) a callee-derives API is supported, allowing AD to be used in a modular fashion; and (6) a path to a formal semantics of AD, and to formal proofs of correctness of systems that use and implement AD, is laid out.

Due to space limitations, the details of how the λ-calculus can be augmented with AD operators is beyond our scope. Instead, we will describe the basic intuitions that underly the approach, and exhibit some preliminary work on its practical benefits. This starts (Section 2) with a discussion of modularity and higher-order functions in a numeric context, where we show how higher-order functions can solve some modularity issues that occur in many current AD systems. We continue (Section 3) by considering the AD transforms as higher-order functions, and in this context we generalize their types. This leads us (Section 4) to note a relationship between the AD operators and the pushforward and pullback constructions of differential geometry, which motivates some details of the types we describe as well as some of the terminology we introduce. In Section 5 we discuss how constructs that appear to the programmer to involve run-time transforms can, by appropriate compiler techniques, be migrated to compile-time. Section 6 describes a system which embodies these principles. It starts with a minimalist language (the λ-calculus augmented with a numeric basis and the AD operators) but uses aggressive compilation techniques to produce object code that is competitive with the most sophisticated current FORTRAN-based AD systems. Armed with this practical benefit, we close (Section 7) with a discussion of other benefits which this new formalism for AD has now put in our reach.

2 Functional Programming and Modularity in AD

Let us consider a few higher-order functions which a numeric programmer might wish to use. Perhaps the most familiar is numeric integration,

```
double nint(double f(double), double x0, double x1);
```

which accepts a function $f : \mathbb{R} \to \mathbb{R}$ and range limits a and b and returns an approximation of $\int_a^b f(x)\,\mathrm{d}x$. In conventional mathematical notation we would say that this function has the type

$$\texttt{nint} : (\mathbb{R} \to \mathbb{R}) \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}.$$

There are a few points we can make about this situation.

First, note that the caller of `nint` might wish to pass an argument function which is not known, at least in its details, until run time. For example, in the straightforward code to evaluate

$$\sum_{i=1}^{n} \int_1^2 (\sin x)^{\cos(x/i)}\,\mathrm{d}x$$

the caller needs to make a function which maps $x \mapsto (\sin x)^{\cos(x/i)}$ for each desired value of i. Although it is possible to code around this necessity by giving `nint` a more complicated API and forcing the caller to package up this extra "environment" information, this is not only cumbersome and error prone but also tends to degrade performance. The notation we will adopt for the construction of a function, "closed" over the values of any relevant variables in scope at the point of creation, is a "λ expression," after which the λ-calculus is named. Here, it would be $(\lambda x\,.\,(\sin x)\hat{}(\cos(x/i)))$.

Second, note that it would be natural to define two-dimensional numeric integration in terms of nested application of `nint`. So for example,

```
double nint2(double f2(double x, double y),
             double x0, double x1,
             double y0, double y1)
{ return nint((λx . nint((λy . f(x,y)), y0, y1)),
              x0, x1); }
```

Third, it turns out that programs written in functional-programming languages are rife with constructs of this sort (for instance, `map` which takes a function and a list and returns a new list whose elements are computed by applying the given function to corresponding elements of the original list); because of this, PLT techniques have been developed to allow compilers for functional languages to optimize across the involved procedure-call barriers. This sort of optimization has implications for numeric programming, as numeric code often calls procedures like `nint` in inner loops. In fact, benchmarks have shown the efficacy of these techniques on numeric code. For instance, code involving a double integral of the sort above experienced an order of magnitude improvement when such techniques were used.

Other numeric routines are also naturally viewed as higher-order functions. Numeric optimization routines, for instance, are naturally formulated as procedures which take the function to be optimized as one argument. In mathematics other concepts are defined as higher-order functions, and if we are to raise the level of expressiveness of scientific programming we might wish to consider using similar conventions when coding such concepts. An enormous number of functions spring to mind: the continuous Fourier transform, or higher-order functions that map differential forms and boundary conditions (each of which might be thought of as a function) to their solution. Even more sophisticated sorts of numeric computations that are difficult to express without the machinery of functional-programming languages, such as pumping methods for increasing rates of convergence, are persuasively discussed elsewhere [4] but stray beyond our present topic.

3 The AD transforms *are* higher-order functions

The first argument `f` to the `nint` procedure of the previous section obeys a particular API: `nint` can call `f`, but (at least in any mainstream language) there are no other operations (with the possible exception of a conservative test for equality) that can be performed on a function passed as an argument. We might imagine improving `nint`'s accuracy and efficiency by having it use derivative information, so that it could more accurately and efficiently adapt its points of evaluation to the local curvature of `f`. Of course, we would want an AD transform of `f` rather than some poor numeric approximation to the desired derivative. Upon deciding to do this, we would have two alternatives. One would be to change the signature of `nint` so that it takes an additional argument `df` that calculates the derivative of `f` at a point. This alternative requires rewriting every call to `nint` to pass this extra argument. Some call sites would be passing a function argument to `nint` that is itself a parameter to the calling routine, resulting in a ripple effect of augmentation of various APIs. This can be seen above, where `nint2` would need to accept an extra parameter—or perhaps two extra parameters. This alternative, which we might call *caller-derives*, requires potentially global changes in order to change a local decision about how a particular numeric integration routine operates, and is therefore a severe violation of the principles of modularity.

The other alternative would be for `nint` to be able to internally find the derivative of `f`, in a *callee-derives* discipline. In order to do this, it would need to be able to invoke AD upon that function argument. To be concrete, we posit two derivative-taking operators which perform the forward- and reverse-mode AD transforms on the functions they are passed.[3] These have a somewhat complex API, so as to avoid repeated calculation of the primal function during derivative calculation. For forward-mode AD, we introduce $\overrightarrow{\mathscr{J}}$ which we for now give a simplified signature $\overrightarrow{\mathscr{J}} : (\mathbb{R}^n \to \mathbb{R}^m) \to ((\mathbb{R}^n \times \mathbb{R}^n) \to (\mathbb{R}^m \times \mathbb{R}^m))$. This takes a numeric function $\mathbb{R}^n \to \mathbb{R}^m$ and returns an augmented function which takes what the original function took along with a perturbation direction in its input space, and returns what the original function returned along with a perturbation direction in its output space. This mapping from an input perturbation to an output perturbation is equivalent to multiplication by the Jacobian. Its reverse-mode AD sibling has a slightly more complex API, which we can caricature as $\overleftarrow{\mathscr{J}} : (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to (\mathbb{R}^m \times (\mathbb{R}^m \to \mathbb{R}^n)))$. This takes a numeric function $\mathbb{R}^n \to \mathbb{R}^m$ and returns an augmented function which takes what the original function took and returns what the original function returned paired with a "reverse phase" function that maps a sensitivity in the output space back to a sensitivity in the input space. This mapping of an output sensitivity to an input sensitivity is equivalent to multiplication by the transpose of the Jacobian.

These AD operators are (however implemented, and whether confined to a pre-processor or supported as dynamic run-time constructs) higher-order functions, but they cannot be written in the conventional λ-calculus. The machinery to allow them to be expressed is somewhat involved [10, 11, 12].

Part of the reason for this complexity can be seen in `nint2` above, which illustrates the need to handle not only anonymous functions but also higher-order functions, nesting, and interactions between variables of various scopes that correspond to the distinct nested invocations of the AD operators. If `nint` is modified to take the derivative of its function argument, then the outer call to `nint` inside `nint2` will take the derivative of an unnamed function which internally invokes `nint`. Since this inner `nint` also invokes the derivative operator, the $\overrightarrow{\mathscr{J}}$ and $\overleftarrow{\mathscr{J}}$ operators must both be able to be applied to functions that internally

[3] One can imagine hybrid operators; we leave that for the future.

invoke $\overrightarrow{\mathscr{J}}$ and $\overleftarrow{\mathscr{J}}$. We also do not wish to introduce a new special "tape" data type onto which computation flow graphs are recorded, as this would both increase the number of data types present in the system, and render the system less amenable to standard optimizations.

Instead, driven by the need to handle nesting and a desire for uniformity, we generalize the AD operators $\overrightarrow{\mathscr{J}}$ and $\overleftarrow{\mathscr{J}}$ to apply not only to numeric functions $\mathbb{R}^n \to \mathbb{R}^m$ but to any function $\alpha \to \beta$, where α and β are arbitrary types. Note that α and β might in fact be function types, so we will be assigning a meaning to "the forward derivative of the higher-order function `map`," or to the derivative of `nint`. This generalization will allow us to mechanically transform the code bodies of functions without regard to the types of the functions called within those code bodies. But in order to understand this generalization, we briefly digress into a mathematical domain that can be used to define and link forward- and reverse-mode AD.

4 AD and Differential Geometry

We now use some concepts from differential geometry to motivate and roughly explain the types and relationships in our λ-calculus augmented with AD operators. However it is important to note that we give a cartoon description here, with many details suppressed or even altered for the sake of brevity, clarity, and intuition.

In differential geometry, a differentiable manifold $\mathscr{N}$ has some structure associated with it. Each point $x \in \mathscr{N}$ has an associated vector space called its tangent space, whose members can be thought of as directions in which x can be locally perturbed in $\mathscr{N}$. We call this a *tangent vector* of x and write it $\overrightarrow{x}$. An element x paired with an element $\overrightarrow{x}$ of the tangent space of x is called a tangent bundle, written $\overrightarrow{x} = (x, \overrightarrow{x})$. A function between two differentiable manifolds, $f : \mathscr{N} \to \mathscr{M}$, which is differentiable at x, mapping it to $y = f(x)$, can be lifted to map *tangent bundles*. In differentiable geometry this is called the pushforward of f. We will write $\overrightarrow{y} = (y, \overrightarrow{y}) = \overrightarrow{f}(\overrightarrow{x}) = \overrightarrow{f}(x, \overrightarrow{x})$. (This notation differs from the usual notation of $T\mathscr{M}_x$ for the tangent space of $x \in \mathscr{M}$.)

We import this machinery of the pushforward, but reinterpret it quite concretely. When f is a function represented in a concrete expression in our augmented λ-calculus, we mechanically transform it into $\overrightarrow{f} = \overrightarrow{\mathscr{J}}(f)$. Moreover when x is a particular value, with a particular shape, we define the shape of $\overrightarrow{x}$, an element of the tangent space of x, in terms of the shape of x. If $x : \alpha$, meaning that x has type (or shape) α, we say that $\overrightarrow{x} : \overrightarrow{\alpha}$ and $\overrightarrow{x} : \overrightarrow{\alpha}$. These proceed by cases, and (with some simplification here for expository purposes) we can say that a perturbation of a real is real, $\overrightarrow{\mathbb{R}} = \mathbb{R}$; the perturbation of a pair is a pair of perturbations, $\overrightarrow{\alpha \times \beta} = \overrightarrow{\alpha} \times \overrightarrow{\beta}$, and the perturbation of a discrete value contains no information, so $\overrightarrow{\alpha} =$ **void** when α is a discrete type like **bool** or **int**. This leaves the most interesting: $\overrightarrow{\alpha \to \beta}$, the perturbation of a function. This is well defined in differential geometry, which would give $\overrightarrow{\alpha \to \beta} = \overrightarrow{\alpha} \to \overrightarrow{\beta}$; but we have an extra complication. We must regard a mapping $f : \alpha \to \beta$ as depending not only on the input value, but also on the value of any free variables that occur in the definition of f. Roughly speaking then, if γ is the type of the combination of all the free variables of the mapping under consideration, which we write as $f : \alpha \xrightarrow{\gamma} \beta$, then $\overrightarrow{\alpha \xrightarrow{\gamma} \beta} = \overrightarrow{\alpha} \xrightarrow{\overrightarrow{\gamma}} \overrightarrow{\beta}$. However we never map such raw tangent values, but always tangent bundles. These have similar signatures, but with tangents always associated with the value whose tangent space they are elements of.

The powerful intuition we now bring from differential geometry is that just as the above allows us to extend the notion of the forward-mode AD transform to arbitrary objects by regarding it as a pushforward of a function defined using the λ-calculus, we can use the notion of a pullback to see how analogous notions can be defined for reverse-mode AD. In differential geometry, a cotangent space is the vector space of linear mappings of elements of the tangent space to reals. We can think of a gradient as a mapping that takes perturbations to reals, where the mapping operation is the dot product. A cotangent is a direct generalization of this notion, giving a sort of generalized gradient. We call the elements of the cotangent space "sensitivities," in keeping with nomenclature from other fields with which the present authors are familiar. However it would also be reasonable to call them adjoint values, as in physics. The cotangent is usually written $T^*\mathscr{M}_x$ but we instead write $\overleftarrow{x} : \overleftarrow{\alpha}$ where $\overleftarrow{x}$ is the sensitivity associated with x and x is of type α. The shape of a sensitivity is defined so that a generalized dot-product operator could be defined, $\bullet : \overleftarrow{\alpha} \times \overrightarrow{\alpha} \to \mathbb{R}$. This induces the types of the sensitivities of functions, so for instance $\overleftarrow{\alpha \xrightarrow{\gamma} \beta} = \overleftarrow{\beta} \to (\overleftarrow{\alpha} \times \overleftarrow{\gamma})$.

The cotangent space in differential geometry is used by the pullback. If $\overrightarrow{f} : (x, \overrightarrow{x}) \mapsto (y, \overrightarrow{y})$ is a pushforward of $f : x \mapsto y$, then the pullback is $\overleftarrow{f} : \overleftarrow{y} \mapsto \overleftarrow{x}$, which must obey the relation $\overleftarrow{y} \bullet \overrightarrow{y} = \overleftarrow{x} \bullet \overrightarrow{x}$. If $\overrightarrow{\mathscr{J}}$ maps functions f to their pushforward $\overrightarrow{f}$, and $\overleftarrow{\mathscr{J}}$ maps functions f to perform the original mapping of f but return the original output of f paired with the pullback of f, then some type simplifications occur. The most important of these is that we can generalize $\overrightarrow{\mathscr{J}}$ and $\overleftarrow{\mathscr{J}}$ to apply not just to *functions* that map between objects of any type, but to apply to *any* object of any type, with functions being a special case: $\overrightarrow{\mathscr{J}} : \alpha \to \overrightarrow{\alpha}$ and $\overleftarrow{\mathscr{J}} : \alpha \to \overleftarrow{\alpha}$. A detailed exposition of this augmented λ-calculus is beyond our scope here. Its definition is a delicate dance, as the new mechanisms must be sufficiently powerful to implement the AD operators, but not so powerful as to preclude their own transformation by AD. We can give however give a bit of a flavor: constructs like $\overrightarrow{\mathscr{J}}(\overleftarrow{\mathscr{J}})$ and its cousins require novel operators like $\overleftarrow{\mathscr{J}}^{-1}$.

5 Migration to Compile Time

In the above exposition, the AD transforms are presented as first-class functions that operate on an even footing with other first-class functions in the system, like `+`. However, compilers are able to migrate many operations that appear to be done at run time to compile time. For instance, the code fragment `(2+3)` might seem to require a run-time addition, but a sufficiently powerful compiler is able to migrate this addition to compile time. A compiler has been constructed, based on the above constructs and ideas, which is able to migrate almost all scaffolding supporting the raw numeric computations to compile time. In essence, a language called VLAD consisting of the above AD mechanisms in addition to a suite of numeric primitives is defined. A compiler for VLAD called STALIN∇ has been constructed (manuscript in review) which uses polyvariant union-free flow analysis. This analysis, for many example programs we have written, allows all scaffolding and function manipulation to be migrated to compile time, leaving for run time a mix of machine instructions whose floating-point density compares favorably to that of code emitted by highly tuned AD systems based on preprocessors and FORTRAN. Although this aggressive compiler currently handles only the forward-mode AD transform, an associated VLAD interpreter handles both the forward- and reverse-mode AD constructs with full general nesting. The compiler is being extended to similarly optimize reverse-mode AD, no significant barriers in this endeavor are anticipated.

Although it is not a production-quality compiler (it is slow, cannot handle large examples, does not support arrays or other update-in-place data structures, and is in general unsuitable for end users) remedying its deficiencies and building a production-quality compiler would be straightforward, involving only known methods [9, 13]. The compiler's limitation to union-free analyses and finite unrolling of recursive data structures could also be relaxed using standard implementation techniques.

6 Some Preliminary Performance Results

We illustrate the power of our techniques with two examples. These were chosen to illustrate a hierarchy of mathematical abstractions built on a higher-order gradient operator. They were *not* chosen to give an advantage to the present system or to compromise performance of other systems. They do however show how awkward it can be to express these concepts in other systems, even overloading-based systems. Variants of these examples were used to exhibit the utility and expressiveness of first-class AD [12].

Figure 1 gives the essence of the two examples. It starts with code shared between these examples: `multivariate-argmin` implements a multivariate optimizer using adaptive naïve gradient descent. This iterates $\mathbf{x}_{i+1} = \eta \nabla f\ \mathbf{x}_i$ until either $\|\nabla f\ \mathbf{x}\|$ or $\|\mathbf{x}_{i+1} - \mathbf{x}_i\|$ is small, increasing η when progress is made and decreasing η when no progress is made. Omitted are definitions for standard SCHEME primitives and the functions `sqr` that squares its argument, `map-n` that maps a function over the list `(0` $\ldots n-1$`)`, `reduce` that folds a binary function with a specified identity over a list, `v+` and `v-` that perform vector addition and subtraction, `k*v` that multiplies a vector by a scalar, `magnitude` that computes the magnitude of a vector,

```
(define ((gradient f) x)
 (let ((n (length x))) ((map-n (lambda (i) (tangent ((j* f) (bundle x (e i n))))) n)))

(define (multivariate-argmin f x)
 (let ((g (gradient f)))
  (letrec ((loop (lambda (x fx gx eta i)
                   (cond ((<= (magnitude gx) (real 1e-5)) x)
                         ((= i (real 10)) (loop x fx gx (* (real 2) eta) (real 0)))
                         (else (let ((x-prime (v- x (k*v eta gx))))
                                 (if (<= (distance x x-prime) (real 1e-5))
                                     x
                                     (let ((fx-prime (f x-prime)))
                                      (if (< fx-prime fx)
                                          (loop x-prime fx-prime (g x-prime) eta (+ i 1))
                                          (loop x fx gx (/ eta (real 2)) (real 0))))))))))))
    (loop x (f x) (g x) (real 1e-5) (real 0)))))

(define (multivariate-argmax f x) (multivariate-argmin (lambda (x) (- (real 0) (f x))) x))

(define (multivariate-max f x) (f (multivariate-argmax f x)))

(define (saddle)
 (let* ((start (list (real 1) (real 1)))
        (f (lambda (x1 y1 x2 y2) (- (+ (sqr x1) (sqr y1)) (+ (sqr x2) (sqr y2)))))
        ((list x1* y1*) (multivariate-argmin
                         (lambda ((list x1 y1)) (multivariate-max
                                                 (lambda ((list x2 y2)) (f x1 y1 x2 y2)) start)) start))
        ((list x2* y2*) (multivariate-argmax (lambda ((list x2 y2)) (f x1* y1* x2 y2)) start)))
  (list (list (write x1*) (write y1*)) (list (write x2*) (write y2*)))))

(define (naive-euler w)
 (let* ((charges (list (list (real 10) (- (real 10) w)) (list (real 10) (real 0))))
        (x-initial (list (real 0) (real 8)))
        (xdot-initial (list (real 0.75) (real 0)))
        (delta-t (real 1e-1))
        (p (lambda (x) ((reduce + (real 0)) ((map (lambda (c) (/ (real 1) (distance x c)))) charges)))))
  (letrec ((loop (lambda (x xdot)
                   (let* ((xddot (k*v (real -1) ((gradient p) x))) (x-new (v+ x (k*v delta-t xdot))))
                    (if (positive? (list-ref x-new 1))
                        (loop x-new (v+ xdot (k*v delta-t xddot)))
                        (let* ((delta-t-f (/ (- (real 0) (list-ref x 1)) (list-ref xdot 1)))
                               (x-t-f (v+ x (k*v delta-t-f xdot))))
                         (sqr (list-ref x-t-f 0))))))))
    (loop x-initial xdot-initial))))

(define (particle)
 (let* ((w0 (real 0)) ((list w*) (multivariate-argmin (lambda ((list w)) (naive-euler w)) (list w0))))
  (write w*)))
```

Fig. 1. The essence of the `saddle` and `particle` examples.

Table 1. Run times of our examples normalized relative to a unit run time for STALIN∇.

	Language/Implementation			
Example	STALIN∇	ADIFOR	TAPENADE	FADBAD++
`saddle`	1.00	0.49	0.72	5.93
`particle`	1.00	0.85	1.76	32.09

`distance` that computes the l^2 norm of the difference of two vectors, and `e` that returns the i-th basis vector of dimension n.

The first example, `saddle`, computes a saddle point: $\min_{(x_1,y_1)} \max_{(x_2,y_2)} ({x_1}^2 + {y_1}^2) - ({x_2}^2 + {y_2}^2)$. The second example, `particle`, models a charged particle traveling non-relativistically in a plane with position $\mathbf{x}(t)$ and velocity $\dot{\mathbf{x}}(t)$ and accelerated by an electric field formed by a pair of repulsive bodies, $p(\mathbf{x};w) = \|\mathbf{x} - (10, 10 - w)\|^{-1} + \|\mathbf{x} - (10, 0)\|^{-1}$, where w is a modifiable control parameter of the system, and hits the x-axis at position $\mathbf{x}(t_f)$. We optimize w so as to minimize $E(w) = x_0(t_f)^2$, with the goal of finding a value for w that causes the particle's path to intersect the origin.

Naïve Euler ODE integration ($\ddot{\mathbf{x}}(t) = -\nabla_{\mathbf{x}} p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)}$; $\dot{\mathbf{x}}(t+\Delta t) = \dot{\mathbf{x}}(t) + \Delta t\, \ddot{\mathbf{x}}(t)$; $\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t)$) is used to compute the particle's path, with a linear interpolation to find the x-axis intersect (when $x_1(t+\Delta t) \leq 0$ we let $\Delta t_f = -x_1(t)/\dot{x}_1(t)$; $t_f = t + \Delta t_f$; $\mathbf{x}(t_f) = \mathbf{x}(t) + \Delta t_f \dot{\mathbf{x}}(t)$ and calculate the final error as $E(w) = x_0(t_f)^2$.) The final error is minimized with respect to w by `multivariate-argmin`.

These examples were chosen because they both illustrate several important characteristics of our compilation techniques. First, they use standard vector arithmetic which, without our techniques, would require allocation and reclamation of new vector objects whose size might be unknown at compile time. Furthermore, access to the components of such vectors would require indirection. Second, they use higher-order functions: ones like `map-n` and `reduce`, that are familiar to the functional-programming community, and ones like `gradient` and `multivariate-argmin`, that are familiar to numeric programmers. Without our techniques, these would require closures and indirect function calls to unspecified targets. Third, they compute nested derivatives, i.e., they take derivatives of functions that take derivatives of other functions. This involves nested application of the AD primitives.

STALIN∇ performed a polyvariant union-free flow analysis on both of these examples, and generated FORTRAN-like code. Variants of these examples were also coded in SCHEME, ML, HASKELL, C++, and FORTRAN, and run with a variety of compilers and AD implementations. Here we discuss only the C++ and FORTRAN versions. For C++, we used the FADBAD++ implementation of forward AD and compiled with G++. For FORTRAN, we used both the ADIFOR and TAPENADE implementations of forward AD and compiled with G77. In all of the variants, we attempted to be faithful to both the generality of the mathematical concepts represented in the examples and to the standard coding style typically used for each particular language. This means in particular that we used "tangent-vector" mode where available, which put STALIN∇ at a disadvantage of about a factor of two from repeated primal computations. (Although STALIN∇ does not implement a tangent-vector mode it would be straightforward to add such a facility.)

Implementing these examples in other AD systems required considerable effort; the details are described in a companion paper. Table 1 summarizes the run times of our examples nor-

malized relative to a unit run time for STALIN∇.[4] This research prototype exhibits an increase in performance of one to three orders of magnitude when compared with the overloading-based forward AD implementations for both functional and imperative languages (of which only the fastest is shown) and roughly matches the performance of the transformation-based forward AD implementations for imperative languages.

7 Discussion and Conclusion

The TAPENADE 2.1 User's Guide [3] Sect. 10 p. 72 states:

> **10. KNOWN PROBLEMS AND DEVELOPMENTS TO COME**
> We conclude this user's guide of TAPENADE by a quick description of known problems, and how we plan to address them in the next releases. [...] we focus on missing functionalities. [...]
> **10.4 Pointers and dynamic allocation**
> Full AD on FORTRAN95 supposes pointer analysis, and an extension of the AD models on programs that use dynamic allocation. This is not done yet.
> Whereas the tangent mode does not pose major problems for programs with pointers and allocation, there are problems in the reverse mode. For example, how should we handle a memory deallocation in the reverse mode? During the reverse sweep, the memory must be reallocated somehow, and the pointers must point back into this reallocated memory. Finding the more efficient way to handle this is still an open problem.

The Future Plans section on the OPENAD web site `http://www-unix.mcs.anl.gov/~utke/OpenAD/` states:

> **4. Language-coverage and library handling in adjoint code**
> **2. language concepts (e.g., array arithmetic, pointers and dynamic memory allocation, polymorphism):**
> Many language concepts, in particular those found in object-oriented languages, have never been considered in the context of automatic adjoint code generation. We are aware of several hard theoretical and technical problems that need to be considered in this context. Without an answer to these open questions the correctness of the adjoint code cannot be guaranteed.

In programming-language theory, semantics are defined by reductions which transform a program from the source language into the λ-calculus, or an equivalent formalism like SSA [1, 6]. Since we have defined the AD operators in a λ-calculus setting in an extremely general fashion, these operators inter-operate correctly with all other constructs in the language. This addresses, in particular, all the above issues, and in fact all such issues: by operating in this framework, the AD constructs become available to the programmer in a dynamic fashion, with extreme generality and uniformity. This framework has another benefit: compiler optimizations and other compiler and implementation techniques are already formulated in the same framework, which allows the AD constructs to be integrated into compilers and combined with aggressive optimization. This gives the numeric programmer the best of both

[4] `http://www.bcl.hamilton.ie/~qobi/tr-08-03/` contains the source code for all variants of our examples, the scripts used to produce Table 1, and the log produced by running those scripts.

worlds: the ability to write confidently in an expressive higher-order modular dynamic style while obtaining competitive numeric performance.

The λ-calculus approach also opens some exciting theoretical questions. The current system is based on the untyped λ-calculus. Can the $\overrightarrow{\mathscr{J}}$ and $\overleftarrow{\mathscr{J}}$ operators be incorporated into a typed λ-calculus? Many models of real computation have been developed; can this system be formalized in that sense? Can the AD operators as defined be proved correct, in the sense of matching a formal specification written in terms of limits or non-intuitive differential geometric constructions? Is there a relationship between this augmented λ-calculus and synthetic differential geometry? Could entire AD systems be built and formally proven correct?

Acknowledgement. This work was supported, in part, by NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and a grant from the Higher Education Authority of Ireland. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

1. Appel, A.W.: SSA is functional programming. ACM SIGPLAN Notices **33**(4), 17–20 (1998)
2. Church, A.: The Calculi of Lambda Conversion. Princeton University Press, Princeton, NJ (1941)
3. Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis (2004). URL `http://www.inria.fr/rrrt/rt-0300.html`
4. Hughes, J.: Why functional programming matters. The Computer Journal **32**(2), 98–107 (1989). URL `http://www.md.chalmers.se/~rjmh/Papers/whyfp.html`
5. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Functional Programming Languages and Computer Architecture. Springer-Verlag, Nancy, France (1985)
6. Kelsey, R.A.: A correspondence between continuation passing style and static single assignment form. ACM SIGPLAN Notices, Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations **30**(3), 13–22 (1995)
7. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part 1. Comm. of the ACM **3**, 184–95 (1960)
8. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.: LISP 1.5 Programmer's Manual. MIT Press, Cambridge, MA (1962). Reprinted Feb. 1965
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, New York (1999)
10. Pearlmutter, B.A., Siskind, J.M.: Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM Trans. on Programming Languages and Systems (2008). To appear
11. Siskind, J.M., Pearlmutter, B.A.: First-class nonstandard interpretations by opening closures. In: Proceedings of the 2007 Symposium on Principles of Programming Languages, pp. 71–6. Nice, France (2007)
12. Siskind, J.M., Pearlmutter, B.A.: Nesting forward-mode AD in a functional framework. Higher-Order and Symbolic Computation (2008). To appear
13. Wadler, P.L.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 61–78. Nice, France (1990)