

# Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator

BARAK A. PEARLMUTTER

Hamilton Institute

and

JEFFREY MARK SISKIND

Purdue University

---

We show how reverse-mode AD (automatic differentiation)—a generalized gradient-calculation operator—can be incorporated as a first-class function in a functional-programming language. An important property of AD transformations is that they preserve certain complexity properties. Here, this property is that the reverse phase of the reverse-mode transform of a function has the same temporal complexity (up to a small constant factor) as the original untransformed function. The main technical difficulty to be faced is that reverse-mode AD must convert fanout (multiple use of a variable) in the untransformed code into addition in the reverse phase of the transformed code. We address this by expressing all straight-line code segments in A-normal form, which makes fanout lexically apparent. Our formulation generalizes reverse-mode AD to apply to arbitrary higher-order functions, while preserving its desirable complexity properties.

Categories and Subject Descriptors: D.3.2.a [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; G.1.4.b [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*Automatic differentiation*

General Terms: Experimentation, Languages, Performance

Additional Key Words and Phrases: closures, compositionality, derivatives, forward-mode AD, function optimization, higher-order functional languages, Jacobian, machine learning, parameter estimation, program transformation, reflection, reverse-mode AD

---

## 1. INTRODUCTION

When you first learned calculus, you learned how to take the derivatives of some simple expressions. Later, you learned the chain rule, the ability to take the derivative of the composition of two functions. The fact that the space of expressions can be defined inductively as a finite set of basis functions closed with function composition, and the fact that you could take the derivative of each basis function as well as function composition, led to two important closure properties. First, you could

---

Author’s address: J. M. Siskind, School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, Room 330, West Lafayette, IN 47907-2035.

Pearlmutter’s work on this paper was supported by Science Foundation Ireland grant 00/PI.1/C067 and the Higher Education Authority of Ireland. Siskind’s work on this paper was supported, in part, by NSF grant CCF-0438806.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20TBD ACM 0164-0925/20TBD/0500-0001 \$5.00

take the derivative of *any* (differentiable) expression. Second, the derivative of an expression was itself an expression. Thus you could take higher-order derivatives.

The traditional method for taking derivatives has a further undesirable property that is often overlooked. The length of the expression denoting the derivative of an expression can be dramatically longer than the length of the original expression. The reason is that  $(uv)' = uv' + u'v$ . Thus the length of the derivative of a product  $u_1 \cdots u_n$  is not linear in  $n$ . Evaluating derivatives could take dramatically more time than evaluating the original expressions. While this may be of little consequence in classical complex analysis, it has practical implications in computational science.

In this paper, we present a method for computing the derivatives of a different space of expressions. We retain the same finite set of basis functions but replace function composition with the lambda calculus. We present a source-to-source transformation for lambda-calculus expressions that plays the same role as the chain rule does for traditional expressions. Doing so leads to three important closure properties. First, like before, our method allows one to take the derivative of *any* (differentiable) lambda-calculus expression. Second, like before, the derivative of a lambda-calculus expression is itself a lambda-calculus expression, allowing one to take higher-order derivatives. Third, unlike before, the length of a transformed lambda-calculus expression is larger than that of the original expression only by a small constant factor. Moreover, the temporal complexity of evaluating a transformed expression is the same as that of the original expression, greater only by a small constant factor.

While there are already extant implementations of forward-mode AD in a functional framework (see Footnote 1), note that computing the gradient of a function  $\mathbb{R}^n \rightarrow \mathbb{R}$  requires  $n$  applications of forward mode and thus introduces an  $O(n)$  slowdown. In contrast, this gradient can be computed with a single application of reverse-mode AD in  $O(1)$ . For this reason, our methods have potentially huge practical application for computational mathematics, where gradients of functions of high-dimensional input, expressed as large complex programs, are needed for tasks like function optimization, function approximation, parameter estimation, and the solution of differential equations.

Our methods are a generalization of a technique known as *Automatic Differentiation* or AD [Griewank, 2000; Corliss et al., 2001]. AD is an established enterprise that seeks to take the derivatives of functions specified as programs through symbolic manipulation rather than finite differencing. AD has traditionally been applied to *imperative* programs in two forms: forward mode [Wengert, 1964; Kedem, 1980] and reverse mode [Speelpenning, 1980; Rall, 1981]. Backpropagation [Rumelhart et al., 1986] is a special case of reverse-mode AD used to compute the gradient of a multi-layer perceptron to minimize an error function when training the weights. The central contribution of this paper is a correct and implemented framework for applying reverse-mode AD to *functional* programs.<sup>1</sup>

---

<sup>1</sup>Forward-mode AD has been previously applied to functional programs in MAPLE [Monagan and Neuenschwander, 1993], HASKELL [Karczmarczuk, 1998a,b, 1999, 2001b], and SCHEME (the SCUTILS package used in Sussman et al., 2001). Our framework also supports applying forward-mode AD to functional programs, incorporating forward mode and reverse mode in a unified fashion that allows them to be applied to each other in the same program. However, with the

Traditional implementations of reverse mode lack the closure property. Derivatives are computed by recording a ‘tape’ of the computation and interpreting (or run-time compiling) a transformation of the tape played back in reverse. This tape is a different kind of entity than the original program. This complicates the process of taking higher-order derivatives. The fact that the tape must be interpreted (or run-time compiled) introduces a slowdown in the process of computing derivatives. In contrast, our method represents the tape as a chain of closures, the same kind of entity as the original program. This simplifies the process of taking higher-order derivatives and makes our approach amenable to efficient code generation with standard compilation techniques for functional-programming languages.

Our method introduces a novel first-class programming-language primitive  $\overleftarrow{\mathcal{J}}$  that performs reverse-mode AD by way of a *global* program transformation. This allows application of the reverse-mode transformation by programs within the language, rather than by a preprocessor. While such transformation is performed reflectively at run time in our prototype implementation (an interpreter), flow analysis and partial evaluation could be used to migrate the transformation to compile time. In the future, we plan to construct such an optimizing compiler for the methods described in this paper using extensions of the techniques from the STALIN compiler for SCHEME [Siskind, 1999].

To achieve closure, our method involves the solution of two technical problems. First, we must support transformation of nested lambda expressions, particularly those with free-variable references. Our method can handle the case where reverse-mode AD is applied to a function  $f$  that takes an argument  $x$  and that, in turn, applies reverse-mode AD to a function  $g$ , nested inside  $f$ , that has a free reference to  $x$ , i.e. the argument to the surrounding function  $f$ . This case is useful, because, as shown in Section 5, it allows computations like  $\min_x \max_y f(x, y)$ , where  $x$  is such a free-variable reference. Second, since to achieve closure it must be possible to apply  $\overleftarrow{\mathcal{J}}$  to *any* function, *inter alia*, we must support application of  $\overleftarrow{\mathcal{J}}$  to itself.<sup>2</sup>

This paper contributes to both the functional programming community and the AD community. To the functional-programming community, it contributes a method for performing AD that has the correct closure and complexity properties. To the AD community, it contributes the ability to apply reverse mode in a nested fashion to closures with free variables.

The methods described below have potential practical application not only in building better functional-programming implementations for scientific computing, but also to building more powerful AD systems for conventional languages. For example, many modern modern FORTRAN compilers use SSA as an intermediate representation, which has been shown to be equivalent to continuation passing style [Kelsey, 1995; Appel, 1998]. Thus, correct efficient nestable reverse-mode AD for the general lambda calculus immediately allows, at least in principle, the incorporation of such

---

exception of the tutorial in Section 2, this paper focuses solely on reverse mode.

<sup>2</sup>An attempt to create a typed lambda calculus incorporating  $\overleftarrow{\mathcal{J}}$  would run into the issue of giving  $\overleftarrow{\mathcal{J}}$  a polymorphic type while allowing self-application like  $(\overleftarrow{\mathcal{J}} \overleftarrow{\mathcal{J}})$ . This same issue crops up with the polymorphic identity function in the conventional typed lambda calculus, although it might be noted that self-application of the identity function arises quite infrequently, while self-application of the AD operators is quite natural, arising from any nested use.

an operator even into a FORTRAN implementation. This has not previously been possible; to date, no AD system for any compiled language has allowed nested use of the reverse-mode AD operator.

The remainder of this paper is organized as follows. Section 2 gives a brief tutorial on AD. Section 3 gives an informal overview of our new method. Section 4 presents the technical details of our method. Section 5 gives examples that illustrate the utility of our method. Section 6 discusses previous work on reverse-mode AD in a functional context. Section 7 discusses fanout and the relationship between fanout, binary functions, and free variables. Section 8 summarized the novel contributions of this paper.

## 2. A BRIEF TUTORIAL ON AD

For the benefit of readers unfamiliar with AD, we give a brief tutorial. Our tutorial will also benefit readers familiar with AD, as we use nonstandard notation and a nonstandard exposition that extends naturally to the later presentation of our method.

In this section and the next, we use  $x$  to denote reals,  $\mathbf{x}$  to denote real vectors,  $\mathbf{X}$  to denote real matrices,  $u$  to denote functions from reals to reals,  $b$  to denote functions from pairs of reals to reals,  $f$  to denote functions from real vectors to real vectors or from real vectors to reals, juxtaposition to denote function application (which we take to associate to the left),  $\mathcal{D}$  to denote the higher-order function that maps functions  $u$  to functions that compute the derivative of  $u$ ,  $\mathcal{D}_1$  and  $\mathcal{D}_2$  to denote the higher-order functions that map functions  $b$  to functions that compute the partial derivatives of  $b$  with respect to their first and second arguments respectively,  $\nabla$  and  $\mathcal{J}$  to denote the higher-order functions that map functions  $f$  to functions that compute the gradient or the Jacobian, respectively, of  $f$  at a real vector,  $\top$  to denote matrix transposition,  $\circ$  to denote function composition,  $+$  to denote either scalar or vector addition, and  $\times$  to denote multiplication of either a matrix times a matrix, a matrix times a vector, a scalar times a vector, or a scalar times a scalar.

A program can be viewed as a composition  $f = f_1 \circ \cdots \circ f_n$ :

$$\begin{aligned} \mathbf{x}_1 &= f_1 \mathbf{x}_0 \\ &\vdots \\ \mathbf{x}_n &= f_n \mathbf{x}_{n-1} \end{aligned}$$

Here, each  $\mathbf{x}_i$  denotes a machine state,  $\mathbf{x}_0$  denotes the input machine state,  $\mathbf{x}_n$  denotes the output machine state, and each  $f_i$  denotes the transition function from machine state  $\mathbf{x}_{i-1}$  to machine state  $\mathbf{x}_i$ . From the chain rule, we have:

$$\begin{aligned} \mathcal{J} f \mathbf{x}_0 &= (\mathcal{J} f_n \mathbf{x}_{n-1}) \times \cdots \times (\mathcal{J} f_1 \mathbf{x}_0) \\ (\mathcal{J} f \mathbf{x}_0)^\top &= (\mathcal{J} f_1 \mathbf{x}_0)^\top \times \cdots \times (\mathcal{J} f_n \mathbf{x}_{n-1})^\top \end{aligned}$$

This leads to two ways to compute the Jacobian of  $f$  at  $\mathbf{x}_0$ :

$$\begin{aligned}\overline{\mathbf{X}}_1 &= (\mathcal{J} f_1 \mathbf{x}_0) \\ \overline{\mathbf{X}}_2 &= (\mathcal{J} f_2 \mathbf{x}_1) \times \overline{\mathbf{X}}_1 \\ &\vdots \\ \overline{\mathbf{X}}_n &= (\mathcal{J} f_n \mathbf{x}_{n-1}) \times \overline{\mathbf{X}}_{n-1}\end{aligned}$$

which computes  $\overline{\mathbf{X}}_n = \mathcal{J} f \mathbf{x}_0$ , and:

$$\begin{aligned}\overleftarrow{\mathbf{X}}_{n-1} &= (\mathcal{J} f_n \mathbf{x}_{n-1})^\top \\ \overleftarrow{\mathbf{X}}_{n-2} &= (\mathcal{J} f_{n-1} \mathbf{x}_{n-2})^\top \times \overleftarrow{\mathbf{X}}_{n-1} \\ &\vdots \\ \overleftarrow{\mathbf{X}}_0 &= (\mathcal{J} f_1 \mathbf{x}_0)^\top \times \overleftarrow{\mathbf{X}}_1\end{aligned}$$

which computes  $\overleftarrow{\mathbf{X}}_0 = (\mathcal{J} f \mathbf{x}_0)^\top$ . These have a downside: storage of the intermediate  $\overline{\mathbf{X}}_i$  and  $\overleftarrow{\mathbf{X}}_i$  variables can be quadratic in the size of the machine state. Furthermore, each requires a special case for the first line. These issues can both be resolved, in the first case, by computing  $\overline{\mathbf{x}}_n = (\mathcal{J} f \mathbf{x}_0) \times \overline{\mathbf{x}}_0$ :

$$\begin{aligned}\overline{\mathbf{x}}_1 &= (\mathcal{J} f_1 \mathbf{x}_0) \times \overline{\mathbf{x}}_0 \\ &\vdots \\ \overline{\mathbf{x}}_n &= (\mathcal{J} f_n \mathbf{x}_{n-1}) \times \overline{\mathbf{x}}_{n-1}\end{aligned}$$

and, in the second case, by computing  $\overleftarrow{\mathbf{x}}_0 = (\mathcal{J} f \mathbf{x}_0)^\top \times \overleftarrow{\mathbf{x}}_n$ :

$$\begin{aligned}\overleftarrow{\mathbf{x}}_{n-1} &= (\mathcal{J} f_n \mathbf{x}_{n-1})^\top \times \overleftarrow{\mathbf{x}}_n \\ &\vdots \\ \overleftarrow{\mathbf{x}}_0 &= (\mathcal{J} f_1 \mathbf{x}_0)^\top \times \overleftarrow{\mathbf{x}}_1\end{aligned}$$

The former is called *forward-mode AD* and the latter is called *reverse-mode AD*. We refer to the  $\mathbf{x}_i$  as the *primal* variables, the  $\overline{\mathbf{x}}_i$  as the *perturbation* variables, and the  $\overleftarrow{\mathbf{x}}_i$  as the *sensitivity* variables. The columns of the Jacobian can be recovered by taking  $\overline{\mathbf{x}}_0$  to be basis vectors, and correspondingly the rows can be recovered by taking  $\overleftarrow{\mathbf{x}}_n$  to be basis vectors.

These perturbation and sensitivity variables can be viewed as follows. The entire program is a function  $f$  which maps  $\mathbf{x}_0$  to  $\mathbf{x}_n$  via intermediate values  $\mathbf{x}_i$ . The perturbation variable  $\overline{\mathbf{x}}_i$  denotes the product of the Jacobian of the function  $f_1 \circ \dots \circ f_i$  evaluated at  $\mathbf{x}_0$  multiplied by  $\overline{\mathbf{x}}_0$ . The sensitivity variable  $\overleftarrow{\mathbf{x}}_i$  denotes the product of the transpose of the Jacobian of the function  $f_{i+1} \circ \dots \circ f_n$  evaluated at  $\mathbf{x}_i$  multiplied by  $\overleftarrow{\mathbf{x}}_n$ .

The transition functions  $f_i$  are typically sparse. They typically compute a single element  $l$  of  $\mathbf{x}_i$ , either as a unary scalar function  $u$  of a single element  $j$  of  $\mathbf{x}_{i-1}$  or

as a binary scalar function  $b$  of two elements  $j$  and  $k$  of  $\mathbf{x}_{i-1}$ , passing the remaining elements of  $\mathbf{x}_{i-1}$  unchanged through to  $\mathbf{x}_i$ . In this case, the Jacobians  $\mathcal{J} f_i \mathbf{x}$  and the products  $(\mathcal{J} f_i \mathbf{x}) \times \overline{\mathbf{x}}$  and  $(\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}}$  are also sparse. In the unary case:

$$\begin{aligned} (f_i \mathbf{x})[j] &= u \mathbf{x}[k] \\ (f_i \mathbf{x})[j'] &= \mathbf{x}[j'] \end{aligned} \quad j' \neq j$$

$$\begin{aligned} (\mathcal{J} f_i \mathbf{x})[j, k] &= \mathcal{D} u \mathbf{x}[k] \\ (\mathcal{J} f_i \mathbf{x})[j', j'] &= 1 \\ (\mathcal{J} f_i \mathbf{x})[j', k'] &= 0 \end{aligned} \quad \begin{array}{l} j' \neq j \\ \text{otherwise} \end{array}$$

$$\begin{aligned} ((\mathcal{J} f_i \mathbf{x}) \times \overline{\mathbf{x}})[j] &= (\mathcal{D} u \mathbf{x}[k]) \times \overline{\mathbf{x}}[k] \\ ((\mathcal{J} f_i \mathbf{x}) \times \overline{\mathbf{x}})[j'] &= \overline{\mathbf{x}}[j'] \end{aligned} \quad j' \neq j$$

$$\begin{aligned} ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[k] &= \overline{\mathbf{x}}[k] + ((\mathcal{D} u \mathbf{x}[k]) \times \overline{\mathbf{x}}[j]) \\ ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[j] &= 0 \\ ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[k'] &= \overline{\mathbf{x}}[k'] \end{aligned} \quad \text{otherwise}$$

In the binary case:

$$\begin{aligned} (f_i \mathbf{x})[j] &= b(\mathbf{x}[k], \mathbf{x}[l]) \\ (f_i \mathbf{x})[j'] &= \mathbf{x}[j'] \end{aligned} \quad j' \neq j$$

$$\begin{aligned} (\mathcal{J} f_i \mathbf{x})[j, k] &= \mathcal{D}_1 b(\mathbf{x}[k], \mathbf{x}[l]) \\ (\mathcal{J} f_i \mathbf{x})[j, l] &= \mathcal{D}_2 b(\mathbf{x}[k], \mathbf{x}[l]) \\ (\mathcal{J} f_i \mathbf{x})[j', j'] &= 1 \\ (\mathcal{J} f_i \mathbf{x})[j', k'] &= 0 \end{aligned} \quad \begin{array}{l} j' \neq j \\ \text{otherwise} \end{array}$$

$$\begin{aligned} ((\mathcal{J} f_i \mathbf{x}) \times \overline{\mathbf{x}})[j] &= ((\mathcal{D}_1 b(\mathbf{x}[k], \mathbf{x}[l])) \times \overline{\mathbf{x}}[k]) + \\ &\quad ((\mathcal{D}_2 b(\mathbf{x}[k], \mathbf{x}[l])) \times \overline{\mathbf{x}}[l]) \\ ((\mathcal{J} f_i \mathbf{x}) \times \overline{\mathbf{x}})[j'] &= \overline{\mathbf{x}}[j'] \end{aligned} \quad j' \neq j$$

$$\begin{aligned} ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[k] &= \overline{\mathbf{x}}[k] + ((\mathcal{D}_1 b(\mathbf{x}[k], \mathbf{x}[l])) \times \overline{\mathbf{x}}[j]) \\ ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[l] &= \overline{\mathbf{x}}[l] + ((\mathcal{D}_2 b(\mathbf{x}[k], \mathbf{x}[l])) \times \overline{\mathbf{x}}[j]) \\ ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[j] &= 0 \\ ((\mathcal{J} f_i \mathbf{x})^\top \times \overline{\mathbf{x}})[k'] &= \overline{\mathbf{x}}[k'] \end{aligned} \quad \text{otherwise}$$

With forward-mode AD, computation of the perturbation variables  $\overline{\mathbf{x}}_i$  can be

ACM Transactions on Programming Languages and Systems, Vol. TBD, No. TBD, TBD 20TBD.

interleaved with the original primal computation:

$$\begin{aligned} \mathbf{x}_1 &= f_1 \mathbf{x}_0 \\ \overline{\mathbf{x}}_1 &= (\mathcal{J} f_1 \mathbf{x}_0) \times \overline{\mathbf{x}}_0 \\ &\vdots \\ \mathbf{x}_n &= f_n \mathbf{x}_{n-1} \\ \overline{\mathbf{x}}_n &= (\mathcal{J} f_n \mathbf{x}_{n-1}) \times \overline{\mathbf{x}}_{n-1} \end{aligned}$$

This leads to a simple transformation:

$$\left. \begin{array}{l} \mathbf{x}_1 = f_1 \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_n = f_n \mathbf{x}_{n-1} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \overrightarrow{\mathbf{x}}_1 = \overrightarrow{f_1} \overrightarrow{\mathbf{x}}_0 \\ \vdots \\ \overrightarrow{\mathbf{x}}_n = \overrightarrow{f_n} \overrightarrow{\mathbf{x}}_{n-1} \end{array} \right.$$

where  $\overrightarrow{\mathbf{x}} = (\mathbf{x}, \overline{\mathbf{x}})$  and  $\overrightarrow{f}(\mathbf{x}, \overline{\mathbf{x}}) = ((f \mathbf{x}), ((\mathcal{J} f \mathbf{x}) \times \overline{\mathbf{x}}))$ . The fact that  $\mathbf{x}_{i-1}$  and  $\overline{\mathbf{x}}_{i-1}$  are no longer referenced once  $\mathbf{x}_i$  and  $\overline{\mathbf{x}}_i$  are computed, coupled with the fact that  $\overrightarrow{\mathbf{x}}$  can be represented as a vector of pairs rather than a pair of vectors, interleaving  $\mathbf{x}$  with  $\overline{\mathbf{x}}$ , means that when the  $f_i$  are sparse, the original program can be written as a sequence of assignments of the form  $x_j := u x_k$  and  $x_j := b(x_k, x_l)$ , referencing variables  $x$  that contain scalars, and the transformed program can be written as a sequence of assignments of the form  $\overrightarrow{x}_j := \overrightarrow{u} \overrightarrow{x}_k$  and  $\overrightarrow{x}_j := \overrightarrow{b}(\overrightarrow{x}_k, \overrightarrow{x}_l)$ , referencing variables  $\overrightarrow{x}$  that hold pairs of  $x$  and  $\overline{x}$ , where:

$$\begin{aligned} \overrightarrow{u}(x, \overline{x}) &= ((u x), ((\mathcal{D} u x) \times \overline{x})) \\ \overrightarrow{b}((x_1, \overline{x}_1), (x_2, \overline{x}_2)) &= ((b(x_1, x_2), \\ &\quad ((\mathcal{D}_1 b(x_1, x_2)) \times \overline{x}_1) + ((\mathcal{D}_2 b(x_1, x_2)) \times \overline{x}_2))) \end{aligned}$$

This means that forward-mode AD can be implemented in almost any programming language and programming style, functional or otherwise, simply by overloading the representation of reals  $x$  with pairs  $\overrightarrow{x}$  of reals  $x$  and  $\overline{x}$  and by overloading the primitives  $u$  and  $b$  with  $\overrightarrow{u}$  and  $\overrightarrow{b}$  respectively. In the functional realm, this has been done in MAPLE [Monagan and Neuenschwander, 1993], HASKELL [Karczmarczuk, 1998a,b, 1999, 2001b], and SCHEME (the SCMUTILS package used in Sussman et al., 2001). Note that forward-mode AD preserves both the temporal and spatial complexity of the program.

In contrast, with reverse-mode AD, computation of the sensitivity variables  $\overline{\mathbf{x}}_i$  *cannot* be interleaved with the original primal computation. The computation must be divided into two phases, a *forward phase* that computes the primal variables and

a *reverse phase* that computes the sensitivity variables in reverse order:

$$\begin{aligned}
 \mathbf{x}_1 &= f_1 \mathbf{x}_0 \\
 &\vdots \\
 \mathbf{x}_n &= f_n \mathbf{x}_{n-1} \\
 \overline{\mathbf{x}}_{n-1} &= (\mathcal{J} f_n \mathbf{x}_{n-1})^\top \times \overline{\mathbf{x}}_n \\
 &\vdots \\
 \overline{\mathbf{x}}_0 &= (\mathcal{J} f_1 \mathbf{x}_0)^\top \times \overline{\mathbf{x}}_1
 \end{aligned}$$

Note that, while, in forward mode,  $\mathbf{x}_i$  is no longer referenced once  $\mathbf{x}_{i+1}$  is computed, in reverse mode, (relevant parts of) the primal variables  $\mathbf{x}_i$  computed during the forward phase must be saved until the corresponding sensitivity variables  $\overline{\mathbf{x}}_i$  are computed during the reverse phase. Also note that, while forward-mode AD can be performed using overloading, a *local* program transformation, the above requires a *global* program transformation.

It is tempting to try to perform reverse-mode AD with a local program transformation:

$$\begin{aligned}
 \mathbf{x}_1 &= f_1 \mathbf{x}_0 \\
 \overline{\mathbf{x}}_1 &= \lambda \overline{\mathbf{x}} \overline{\mathbf{x}}_0 ((\mathcal{J} f_1 \mathbf{x}_0)^\top \times \overline{\mathbf{x}}) \\
 &\vdots \\
 \mathbf{x}_n &= f_n \mathbf{x}_{n-1} \\
 \overline{\mathbf{x}}_n &= \lambda \overline{\mathbf{x}} \overline{\mathbf{x}}_{n-1} ((\mathcal{J} f_n \mathbf{x}_{n-1})^\top \times \overline{\mathbf{x}})
 \end{aligned}$$

If we take  $\overline{\mathbf{x}}_0$  to be the identity function, the reverse phase can be performed by evaluating  $\overline{\mathbf{x}}_n \overline{\mathbf{x}}_n$ . We refer to  $\overline{\mathbf{x}}$  as a *backpropagator* variable. Note that each backpropagator variable  $\overline{\mathbf{x}}_i$  closes over the previous backpropagator variable  $\overline{\mathbf{x}}_{i-1}$  to implement sequencing of the reverse phase. Also note that each backpropagator variable  $\overline{\mathbf{x}}_i$  also closes over the corresponding previous primal variable  $\mathbf{x}_{i-1}$  to preserve the necessary values until the reverse phase. This leads to a simple transformation:

$$\left. \begin{aligned} \mathbf{x}_1 &= f_1 \mathbf{x}_0 \\ &\vdots \\ \mathbf{x}_n &= f_n \mathbf{x}_{n-1} \end{aligned} \right\} \rightsquigarrow \left\{ \begin{aligned} \overleftarrow{\mathbf{x}}_1 &= \overleftarrow{f}_1 \overleftarrow{\mathbf{x}}_0 \\ &\vdots \\ \overleftarrow{\mathbf{x}}_n &= \overleftarrow{f}_n \overleftarrow{\mathbf{x}}_{n-1} \end{aligned} \right.$$

where  $\overleftarrow{\mathbf{x}} = (\mathbf{x}, \overline{\mathbf{x}})$  and  $\overleftarrow{f}(\mathbf{x}, \overline{\mathbf{x}}) = ((f \mathbf{x}), (\lambda \overline{\mathbf{x}} \overline{\mathbf{x}} ((\mathcal{J} f \mathbf{x})^\top \times \overline{\mathbf{x}})))$ . However, unlike in forward mode, where  $\overrightarrow{\mathbf{x}} = (\mathbf{x}, \overline{\mathbf{x}})$  can be interleaved as a vector of pairs, it is not possible to interleave  $\overleftarrow{\mathbf{x}} = (\mathbf{x}, \overline{\mathbf{x}})$ , because  $\overline{\mathbf{x}}$  is a function, not a vector. Thus, one must use different techniques to implement reverse-mode AD with a local program transformation that takes advantage of sparsity.

The traditional way this is done is to maintain a single global backpropagator

variable  $\overline{\mathbf{x}}$  that is updated via side effect and by taking:

$$\overleftarrow{f} \mathbf{x} = \mathbf{begin} \overline{\mathbf{x}} := \lambda \overline{\mathbf{x}} \overline{\mathbf{x}} ((\mathcal{J} f \mathbf{x})^\top \times \overline{\mathbf{x}}); \\ (f \mathbf{x}) \mathbf{end}$$

This eliminates the need to pair backpropagators with primal values and allows taking  $\overleftarrow{\mathbf{x}} = \mathbf{x}$ . When the  $f_i$  are sparse, and the original program is written as a sequence of assignments of the form  $x_j := u x_k$  and  $x_j := b(x_k, x_l)$ , referencing variables  $x$  that contain scalars, the transformed program can be written as a sequence of assignments<sup>3</sup> of the form:

$$\overline{x} := \lambda [] \mathbf{begin} \overline{x_j} := 0; \\ \overline{x_k} += (\mathcal{D} u \overleftarrow{x_k}) \times \overline{x_j}; \\ \overline{x} [] \mathbf{end} \\ \overleftarrow{x_j} := u \overleftarrow{x_k}$$

and:

$$\overline{x} := \lambda [] \mathbf{begin} \overline{x_j} := 0; \\ \overline{x_k} += (\mathcal{D}_1 b(\overleftarrow{x_k}, \overleftarrow{x_l})) \times \overline{x_j}; \\ \overline{x_l} += (\mathcal{D}_2 b(\overleftarrow{x_k}, \overleftarrow{x_l})) \times \overline{x_j}; \\ \overline{x} [] \mathbf{end} \\ \overleftarrow{x_j} := b(\overleftarrow{x_k}, \overleftarrow{x_l})$$

taking  $\overleftarrow{x} = x$  and  $x += e$  to denote  $x := x + e$ .

Traditional implementations refer to  $\overline{x}$  as the *tape*, usually implemented as an interpreted (or run-time-compiled) data structure rather than as a chain of closures. For straight-line code, one can dispense with the tape if one admits a global program transformation. One simply postpends the program with assignments to initialize the sensitivity variables and then postpends assignments of the form:

$$\overline{x_j} := 0 \\ \overline{x_k} += (\mathcal{D} u \overleftarrow{x_k}) \times \overline{x_j}$$

for each primal assignment  $x_j := u x_k$ , and of the form:

$$\overline{x_j} := 0 \\ \overline{x_k} += (\mathcal{D}_1 b(\overleftarrow{x_k}, \overleftarrow{x_l})) \times \overline{x_j} \\ \overline{x_l} += (\mathcal{D}_2 b(\overleftarrow{x_k}, \overleftarrow{x_l})) \times \overline{x_j}$$

<sup>3</sup>We are deliberately imprecise here as to the semantics of assignment in the presence of closures. The intent is to close over the current value of a variable and have the closed-over value remain unchanged when a variable is mutated. Note that the backpropagators take no argument and return no result. They are executed for side effect. The reverse phase is performed by appropriately initializing all output sensitivity variables to the values of  $\overline{\mathbf{x}_n}$ , initializing all other sensitivity variables to zero, calling the backpropagator  $\overline{x}$ , and examining the values of  $\overline{\mathbf{x}_0}$  that remain in select sensitivity variables after the backpropagator returns.

for each primal assignment  $x_j := b(x_k, x_l)$ , in reverse order to their occurrence in the primal.

Note that reverse-mode AD preserves the temporal complexity of the program, but not its spatial complexity, due to the need to save primal values for use during the reverse phase. Also, while this approach can be implemented as a local program transformation in most programming languages, it is not amenable to a functional style due to the use of side effects.

### 3. AN INFORMAL OVERVIEW OF OUR METHOD

We have developed a novel method for performing reverse-mode AD in a functional framework. In this section, we present an informal overview of this method. We do this by way of a small running example. We present the technical details of our method in the next section.

First consider a restricted straight-line program that operates on real-valued variables  $x$  with unary functions  $u$  from reals to reals, taking  $x_0$  as the input and producing  $x_n$  as the output:

$$\begin{aligned} x_{j_1} &:= u_1 x_{k_1} \\ &\vdots \\ x_{j_n} &:= u_n x_{k_n} \end{aligned}$$

From Section 2, the tapeless global reverse-mode transformation of this program is:

$$\left. \begin{array}{l} x_{j_1} := u_1 x_{k_1} \\ \vdots \\ x_{j_n} := u_n x_{k_n} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} x_{j_1} := u_1 x_{k_1} \\ \vdots \\ x_{j_n} := u_n x_{k_n} \\ \overline{x_0} := 0 \\ \vdots \\ \overline{x_{n-1}} := 0 \\ \overline{x_{j_n}} := 0 \\ \overline{x_{k_n}} + := (\mathcal{D} u_n x_{k_n}) \times \overline{x_{j_n}} \\ \vdots \\ \overline{x_{j_1}} := 0 \\ \overline{x_{k_1}} + := (\mathcal{D} u_1 x_{k_1}) \times \overline{x_{j_1}} \end{array} \right.$$

If we restrict our consideration to single-assignment code, the assignments  $\overline{x_{j_i}} := 0$

during the reverse phase can be eliminated:

$$\left. \begin{array}{l} x_1 = u_1 x_{k_1} \\ \vdots \\ x_n = u_n x_{k_n} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} x_1 = u_1 x_{k_1} \\ \vdots \\ x_n = u_n x_{k_n} \\ \overline{x_0} := 0 \\ \vdots \\ \overline{x_{n-1}} := 0 \\ \overline{x_{k_n}} + := (\mathcal{D} u_n x_{k_n}) \times \overline{x_n} \\ \vdots \\ \overline{x_{k_1}} + := (\mathcal{D} u_1 x_{k_1}) \times \overline{x_1} \end{array} \right.$$

If we take  $\overline{u_i} = \lambda \overline{x} (\mathcal{D} u_i x_{k_i}) \times \overline{x}$ , this gives:

$$\left. \begin{array}{l} x_1 = u_1 x_{k_1} \\ \vdots \\ x_n = u_n x_{k_n} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} x_1 = u_1 x_{k_1} \\ \vdots \\ x_n = u_n x_{k_n} \\ \overline{x_0} := 0 \\ \vdots \\ \overline{x_{n-1}} := 0 \\ \overline{x_{k_n}} + := \overline{u_n} \overline{x_n} \\ \vdots \\ \overline{x_{k_1}} + := \overline{u_1} \overline{x_1} \end{array} \right.$$

If we further take  $\overleftarrow{u} x = ((u x), (\lambda \overline{x} (\mathcal{D} u x) \times \overline{x}))$ , we can write:

$$\left. \begin{array}{l} x_1 = u_1 x_{k_1} \\ \vdots \\ x_n = u_n x_{k_n} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} (x_1, \overline{x_1}) = \overleftarrow{u_1} x_{k_1} \\ \vdots \\ (x_n, \overline{x_n}) = \overleftarrow{u_n} x_{k_n} \\ \overline{x_0} := 0 \\ \vdots \\ \overline{x_{n-1}} := 0 \\ \overline{x_{k_n}} + := \overline{x_n} \overline{x_n} \\ \vdots \\ \overline{x_{k_1}} + := \overline{x_1} \overline{x_1} \end{array} \right.$$

Note that this transformation is valid only for single-assignment code. The backpropagator variables  $\overleftarrow{x}_i$  are accessed during the reverse phase in reverse order to which they were assigned during the forward phase. Applying this transformation to non-single-assignment code would result in the backpropagators being overwritten during the forward phase and the wrong backpropagators being called during the reverse phase. While the output of  $\overleftarrow{u}$  is a pair consisting of the primal value and a backpropagator, the input is solely a primal, and each  $\overleftarrow{x}_i$  is simply a function from  $\overleftarrow{x}_i$  to  $\overleftarrow{x}_{k_i}$ . Evaluating  $\overleftarrow{x}_i \overleftarrow{x}_i$  has the same temporal complexity as evaluating  $u_i x_{j_i}$ . This is the key property that leads to our method having the appropriate temporal complexity.

Let us now assume that the primitives  $u$  are stored in variables  $x$  and that the reverse-transformed primitives  $\overleftarrow{u}$  are also stored in variables  $\overleftarrow{x}$ . In the untransformed program, a variable  $x$  can contain either a real value or a primitive. For the sake of symmetry, we will construct the transformed program out of corresponding transformed variables  $\overleftarrow{x}$  that can contain either *transformed real values* or transformed primitives. For reasons that we will discuss in Section 4, transformed real values are simply tagged real values. Transformed primitives will map transformed real values to transformed real values paired with backpropagators. This leads to the following transformation:

$$\left. \begin{array}{l} x_1 = x_{j_1} x_{k_1} \\ \vdots \\ x_n = x_{j_n} x_{k_n} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} (\overleftarrow{x}_1, \overleftarrow{x}_1) = \overleftarrow{x}_{j_1} \overleftarrow{x}_{k_1} \\ \vdots \\ (\overleftarrow{x}_n, \overleftarrow{x}_n) = \overleftarrow{x}_{j_n} \overleftarrow{x}_{k_n} \\ \overleftarrow{x}_0 := 0 \\ \vdots \\ \overleftarrow{x}_{n-1} := 0 \\ \overleftarrow{x}_{k_n} + := \overleftarrow{x}_n \overleftarrow{x}_n \\ \vdots \\ \overleftarrow{x}_{k_1} + := \overleftarrow{x}_1 \overleftarrow{x}_1 \end{array} \right.$$

Let us generalize further by allowing the variables  $x$  in the untransformed program to contain arbitrary programming-language values and the primitives in the untransformed program to map arbitrary values to arbitrary values. Doing so requires us to generalize transformed variables  $\overleftarrow{x}$  and sensitivity variables  $\overleftarrow{x}$  to contain arbitrary transformed and sensitivity values that correspond to or match the arbitrary untransformed values stored in the corresponding variables  $x$ . This requires us to add arbitrary sensitivity values. We postulate a function  $\oplus$  to do so. The function  $\oplus$  takes two conformant values of the same shape and returns a result of the same shape. This also requires us to initialize the sensitivity variables with matching zeros. We postulate a function  $\overleftarrow{\mathcal{J}}$  that maps values to corresponding transformed values, the inverse function  $\overleftarrow{\mathcal{J}}^{-1}$  that maps transformed values to corresponding untransformed values, and a function  $\mathbf{0}$  that maps values to matching

zeros. Thus  $\overleftarrow{x}$  is initialized to  $\mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x})$ :

$$\left. \begin{array}{l} x_1 = x_{i_1} x_{j_1} \\ \vdots \\ x_n = x_{i_n} x_{j_n} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} (\overleftarrow{x_1}, \overleftarrow{x_1}) = \overleftarrow{x_{i_1}} \overleftarrow{x_{j_1}} \\ \vdots \\ (\overleftarrow{x_n}, \overleftarrow{x_n}) = \overleftarrow{x_{i_n}} \overleftarrow{x_{j_n}} \\ \overleftarrow{x_0} := \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_0}) \\ \vdots \\ \overleftarrow{x_{n-1}} := \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_{n-1}}) \\ \overleftarrow{x_{j_n}} \oplus := \overleftarrow{x_n} \overleftarrow{x_n} \\ \vdots \\ \overleftarrow{x_{j_1}} \oplus := \overleftarrow{x_1} \overleftarrow{x_1} \end{array} \right.$$

The above transformation of single-assignment straight-line code can be applied to transform any  $\alpha$ -converted lambda expression in A-normal form [Sabry and Felleisen, 1993]. (The requirement for  $\alpha$ -conversion comes from the same underlying constraint as the need for the straight-line code to be single-assignment.) Note that the forward and reverse phases are separated. The forward phase returns a transformed value paired with a function that performs the reverse phase. This function maps  $\overleftarrow{x_n}$  to  $\overleftarrow{x_0}$ , by multiplying the transpose of the Jacobian of the function that maps  $x_0$  to  $x_n$ , at  $x_0$ , by  $\overleftarrow{x_n}$ , under appropriate generalizations of the notions of vectors, matrices, Jacobians, matrix transposition, and matrix-vector multiplication to data of arbitrary shape. It can thus be viewed as a backpropagator. We now have a self-similarity property whereby transformed primitives and transformed lambda expressions both map transformed values to transformed values paired with backpropagators. Thus untransformed and transformed code can treat primitive and user-defined functions equivalently:

$$\left. \begin{array}{l} \lambda x_0 \text{ let } x_1 \triangleq x_{i_1} x_{j_1}; \\ \vdots \\ x_n \triangleq x_{i_n} x_{j_n} \\ \text{in } x_n \text{ end} \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \lambda \overleftarrow{x_0} \text{ let } (\overleftarrow{x_1}, \overleftarrow{x_1}) \triangleq \overleftarrow{x_{i_1}} \overleftarrow{x_{j_1}}; \\ \vdots \\ (\overleftarrow{x_n}, \overleftarrow{x_n}) \triangleq \overleftarrow{x_{i_n}} \overleftarrow{x_{j_n}} \\ \text{in } (\overleftarrow{x_n}, (\lambda \overleftarrow{x_n} \text{ let } \overleftarrow{x_0} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_0}); \\ \vdots \\ \overleftarrow{x_{n-1}} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_{n-1}}); \\ \overleftarrow{x_{j_n}} \oplus \triangleq \overleftarrow{x_n} \overleftarrow{x_n}; \\ \vdots \\ \overleftarrow{x_{j_1}} \oplus \triangleq \overleftarrow{x_1} \overleftarrow{x_1} \\ \text{in } \overleftarrow{x_0} \text{ end})) \text{ end} \end{array} \right.$$

The above formulation, however, does not support nested lambda expressions. The difficulty in supporting nested lambda expressions, and in particular free variables,

is illustrated by the following example. Consider a function  $f \triangleq \lambda a ((\lambda b \lambda c b) a) 1$ . Since  $f$  is the identity function, its derivative is the constant function one. Converting  $f$  to A-normal form gives:

$$\begin{aligned}
 f &\triangleq \lambda a \text{ let } x_1 \triangleq \lambda b \text{ let } x_4 \triangleq \lambda c b \\
 &\quad \text{in } x_4 \text{ end;} \\
 x_2 &\triangleq x_1 a; \\
 x_3 &\triangleq x_2 1 \quad /*a*/ \\
 &\text{in } x_3 \text{ end}
 \end{aligned}$$

If we attempt to transform  $f$  using the above method we get:

$$\begin{aligned}
 \overleftarrow{f} &\triangleq \lambda \overleftarrow{a} \text{ let } \overleftarrow{x_1} \triangleq \lambda \overleftarrow{b} \text{ let } \overleftarrow{x_4} \triangleq \lambda \overleftarrow{c} (\overleftarrow{b}, (\lambda \overleftarrow{b} \overleftarrow{c})) /*b*/ \\
 &\quad \text{in } (\overleftarrow{x_4}, (\lambda \overleftarrow{x_4} \text{ let } \overleftarrow{b} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{b}) \\
 &\quad \quad \text{in } \overleftarrow{b} \text{ end})) \text{ end;} \\
 (\overleftarrow{x_2}, \overleftarrow{x_2}) &\triangleq \overleftarrow{x_1} \overleftarrow{a}; \\
 (\overleftarrow{x_3}, \overleftarrow{x_3}) &\triangleq \overleftarrow{x_2} \overleftarrow{1} \quad /*c*/ \\
 \text{in } (\overleftarrow{x_3}, (\lambda \overleftarrow{x_3} \text{ let } \overleftarrow{a} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{a}); \\
 &\quad \overleftarrow{x_1} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_1}); \\
 &\quad \overleftarrow{x_2} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_2}); \\
 &\quad \overleftarrow{1} \oplus \triangleq \overleftarrow{x_3} \overleftarrow{x_3}; \quad /*d*/ \\
 &\quad \overleftarrow{a} \oplus \triangleq \overleftarrow{x_2} \overleftarrow{x_2} \\
 &\quad \text{in } \overleftarrow{a} \text{ end})) \text{ end}
 \end{aligned}$$

The above code is trivially incorrect, because there are references to unbound variables  $\overleftarrow{c}$ ,  $\overleftarrow{1}$ , and  $\overleftarrow{1}$  in lines  $b$ ,  $c$ , and  $d$ . The free reference to  $\overleftarrow{1}$  in line  $c$  results from transforming the constant 1 in line  $a$  of the untransformed code for  $f$ . We can treat such constants as free references to variables bound in the environment over which a function is closed. When we transform such a closure, we will need to transform the variables and values in its environment. This legitimizes the free reference to  $\overleftarrow{1}$  in line  $c$  but does not address the free references to  $\overleftarrow{c}$  and  $\overleftarrow{1}$  in lines  $b$  and  $d$ . We solve this problem by generating bindings, in the backpropagator for a transformed lambda expression, for all of the sensitivity variables that correspond to free variables in the untransformed lambda expression, that initialize those sensitivity variables to zeros. This is done for  $\overleftarrow{c}$  and  $\overleftarrow{1}$  in lines  $e$  and  $f$

below:

$$\begin{aligned}
\overleftarrow{f} &\triangleq \lambda \overleftarrow{a} \text{ let } \overleftarrow{x_1} \triangleq \lambda \overleftarrow{b} \text{ let } \overleftarrow{x_4} \triangleq \lambda \overleftarrow{c} (\overleftarrow{b}, (\lambda \overleftarrow{b} \text{ let } \overleftarrow{c} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{c}) /*e*/ \\
&\quad \text{in } \overleftarrow{c} \text{ end})) \\
&\quad \text{in } (\overleftarrow{x_4}, (\lambda \overleftarrow{x_4} \text{ let } \overleftarrow{b} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{b}) \\
&\quad \quad \text{in } \overleftarrow{b} \text{ end})) \text{ end}; \\
(\overleftarrow{x_2}, \overleftarrow{x_2}) &\triangleq \overleftarrow{x_1} \overleftarrow{a}; \\
(\overleftarrow{x_3}, \overleftarrow{x_3}) &\triangleq \overleftarrow{x_2} \overleftarrow{1} \\
\text{in } (\overleftarrow{x_3}, (\lambda \overleftarrow{x_3} \text{ let } \overleftarrow{a} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{a}); \\
&\quad \overleftarrow{x_1} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_1}); \\
&\quad \overleftarrow{x_2} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_2}); \\
&\quad \overleftarrow{1} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{1}); /*f*/ \\
&\quad \overleftarrow{1} \oplus \triangleq \overleftarrow{x_3} \overleftarrow{x_3}; \\
&\quad \overleftarrow{a} \oplus \triangleq \overleftarrow{x_2} \overleftarrow{x_2} \\
&\quad \text{in } \overleftarrow{a} \text{ end})) \text{ end}
\end{aligned}$$

Now  $\overleftarrow{f}$  is syntactically correct. Unfortunately, however, it produces the wrong result. If we apply  $\overleftarrow{f}$  to  $\overleftarrow{4}$  we get  $\overleftarrow{4}$  (the correct answer) paired with a backpropagator. But if we call that backpropagator on 1 we get 0 when we should get 1, namely the derivative of  $f$  at 4. To see why, we can trace through the evaluation of the backpropagator. First,  $\overleftarrow{x_3}$  is bound to 1. Then,  $\overleftarrow{a}$ ,  $\overleftarrow{x_1}$ ,  $\overleftarrow{x_2}$ , and  $\overleftarrow{1}$  are bound to zeros. Then, we apply  $\overleftarrow{x_3}$  to 1. Since  $\overleftarrow{x_3}$  is bound to  $\lambda \overleftarrow{b} \dots$ ,  $\overleftarrow{b}$  is bound to 1,  $\overleftarrow{c}$  is bound to a zero,  $\lambda \overleftarrow{b} \dots$  returns a zero, and  $\overleftarrow{1}$  is incremented by a zero and remains a zero. Then, we apply  $\overleftarrow{x_2}$  to a zero. Since  $\overleftarrow{x_2}$  is bound to  $\lambda \overleftarrow{x_4} \dots$ ,  $\overleftarrow{x_4}$  is bound to a zero,  $\overleftarrow{b}$  is bound to a zero,  $\lambda \overleftarrow{x_4} \dots$  returns a zero, and  $\overleftarrow{a}$  is incremented by a zero and remains a zero. This zero is then returned.

The problem results from the fact that the output of the function  $\lambda c b$  in the untransformed  $f$  does not depend on its input. Instead, it depends on the value of a free variable that is the input to the surrounding function  $\lambda b \lambda c b$ . So far, our backpropagators only propagate sensitivities from function outputs to their inputs. They do not propagate sensitivities to the environments over which they are closed.

This problem can be solved by making three changes to the above formulation. First, backpropagators are modified so that instead of having them map output sensitivities to input sensitivities, they map output sensitivities to pairs containing both the environment sensitivities and the input sensitivities, as shown in lines  $g$ ,  $i$ , and  $m$  below. Environment sensitivities are represented as lists of the sensitivities of all of the free variables. Second, the lines in backpropagators that correspond to applications in the untransformed function are modified to accumulate the paired backpropagator result into the sensitivity of the target paired with the sensitivity of its argument, as shown in lines  $j$  and  $k$  below. Finally, lines are generated in backpropagators that correspond to nested lambda expressions in the untransformed

function, as shown in lines  $h$  and  $l$  below:

$$\begin{aligned}
\overleftarrow{f} \triangleq \lambda \overleftarrow{a} \text{ let } \overleftarrow{x_1} \triangleq \lambda \overleftarrow{b} \text{ let } \overleftarrow{x_4} \triangleq \lambda \overleftarrow{c} \text{ (} \overleftarrow{b}, (\lambda \overleftarrow{b} \text{ let } \overleftarrow{c} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{c}) \\
& \text{in } ([\overleftarrow{b}], \overleftarrow{c}) \text{ end}))} \quad /*g*/ \\
& \text{in } (\overleftarrow{x_4}, (\lambda \overleftarrow{x_4} \text{ let } \overleftarrow{b} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{b}); \\
& \quad [\overleftarrow{b}] \oplus \triangleq \overleftarrow{x_4} \quad /*h*/ \\
& \text{in } ([], \overleftarrow{b}) \text{ end})) \text{ end;} \quad /*i*/ \\
(\overleftarrow{x_2}, \overleftarrow{x_2}) \triangleq \overleftarrow{x_1} \overleftarrow{a}; \\
(\overleftarrow{x_3}, \overleftarrow{x_3}) \triangleq \overleftarrow{x_2} \overleftarrow{1} \\
\text{in } (\overleftarrow{x_3}, (\lambda \overleftarrow{x_3} \text{ let } \overleftarrow{a} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{a}); \\
& \quad \overleftarrow{x_1} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_1}); \\
& \quad \overleftarrow{x_2} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{x_2}); \\
& \quad \overleftarrow{1} \triangleq \mathbf{0} (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{1}); \\
& \quad (\overleftarrow{x_2}, \overleftarrow{1}) \oplus \triangleq \overleftarrow{x_3} \overleftarrow{x_3}; \quad /*j*/ \\
& \quad (\overleftarrow{x_1}, \overleftarrow{a}) \oplus \triangleq \overleftarrow{x_2} \overleftarrow{x_2}; \quad /*k*/ \\
& \quad [] \oplus \triangleq \overleftarrow{x_1} \quad /*l*/ \\
& \text{in } ([], \overleftarrow{a}) \text{ end})) \text{ end} \quad /*m*/
\end{aligned}$$

To see how this works, we trace through the evaluation of this new backpropagator. First,  $\overleftarrow{x_3}$  is bound to 1. Then,  $\overleftarrow{a}$ ,  $\overleftarrow{x_1}$ ,  $\overleftarrow{x_2}$ , and  $\overleftarrow{1}$  are bound to zeros. Then, we apply  $\overleftarrow{x_3}$  to 1. Since  $\overleftarrow{x_3}$  is bound to  $\lambda \overleftarrow{b} \dots$ ,  $\overleftarrow{b}$  is bound to 1 and  $\overleftarrow{c}$  is bound to a zero. So far, the evaluation is the same as before. Now we see the first difference. The function  $\lambda \overleftarrow{b} \dots$  returns [1] paired with a zero,  $\overleftarrow{x_2}$  is incremented by [1] to become [1], and  $\overleftarrow{1}$  is incremented by a zero and remains a zero. Then, we apply  $\overleftarrow{x_2}$  to [1]. Since  $\overleftarrow{x_2}$  is bound to  $\lambda \overleftarrow{x_4} \dots$ ,  $\overleftarrow{x_4}$  is bound to [1] and  $\overleftarrow{b}$  is bound to a zero. Then  $[\overleftarrow{b}]$  is incremented by [1]. This increments  $\overleftarrow{b}$  by 1, allowing  $\lambda \overleftarrow{x_4} \dots$  to return [] paired with 1. The variable  $\overleftarrow{x_1}$  is then incremented by [] and  $\overleftarrow{a}$  is incremented by 1 to become 1. This 1 is then returned.

Several subtle issues must be addressed to flesh out this method. First, lambda expressions may have multiple free variables. Thus the lists of sensitivities to these variables, as in lines  $g$ ,  $h$ ,  $i$ ,  $l$ , and  $m$  above, could contain multiple values. Since these lists of sensitivities must conform to be added by  $\oplus$ , we need to adopt a canonical order to the elements of these lists. This is done by assuming a total order on all variables. Second, while the input to this transformation is an  $\alpha$ -converted expression in A-normal form, the output is not. To allow repeated application of this transformation, the output of the transformation must subsequently be  $\alpha$ -converted and converted to A-normal form. Such repeated transformation can yield multiply-transformed variables like  $\overleftarrow{\overleftarrow{x}}$  that are bound to multiply-transformed values. Third, a transformed function can have free variables that do not correspond to free variables in the untransformed function. This is because the transformation introduces references to functions like  $\mathbf{0}$  and  $\overleftarrow{\mathcal{J}}^{-1}$  that may not be used in the untransformed function. (And even if they were, they would be transformed, but the transformed function needs to access the untransformed variants as well.) Thus the environment

sensitivity of a transformed function will be of a different shape than the environment sensitivity of its corresponding untransformed function. For reasons beyond the scope of this paper, we wish the environment sensitivity of a transformed function to be of the same shape as the environment sensitivity of its corresponding untransformed function. To accomplish this, we adopt the convention that environment sensitivities of potentially multiply-transformed functions only contain entries that correspond to free variables in the original completely-untransformed function. We refer to such variables as *base free variables*.

#### 4. THE TECHNICAL DETAILS OF OUR METHOD

Since our method involves a global program transformation, we wish to make this transformation available as a first-class programming-language function  $\overleftarrow{\mathcal{J}}$ . This allows application of this transformation by programs within the language, rather than by a preprocessor. Since  $\overleftarrow{\mathcal{J}}$  must have the ability to reflectively access and transform expressions associated with closures, it is not possible to implement  $\overleftarrow{\mathcal{J}}$  as code within a language that lacks the capacity for such reflection. In such languages,  $\overleftarrow{\mathcal{J}}$  must be added to the language implementation as a new primitive. While it is possible to do this for an existing implementation of an existing language, so long as that implementation internally provides the ability for such reflection, to simplify presentation and experimentation, we formulate the ideas in this paper within a minimalist functional language called VLAD<sup>4</sup> and a minimalist implementation of VLAD called STALIN $\nabla$  (pronounced Stalingrad).<sup>5</sup> VLAD and STALIN $\nabla$  support both forward-mode and reverse-mode AD, but in this paper we only describe reverse mode. VLAD and STALIN $\nabla$ , however, are simply expedient vehicles for exposition and research. The  $\overleftarrow{\mathcal{J}}$  primitive could be added to an existing implementation of an existing language, albeit with considerably greater effort, so long as that implementation internally provides the ability for the necessary reflection.

VLAD is similar to SCHEME. The important differences are summarized below:

- Only functional (side-effect free) constructs are supported.
- The only data types supported are the empty list, Booleans, real numbers, pairs, and functions that take one argument and return one result.
- Since all functions, including primitives, take one argument, those that naturally take multiple arguments (except for `cons` and `list`) take those arguments as tuples constructed from pairs.
- The `cons` and `list` constructs are syntax that expand into curried calls.
- The syntax of lambda expressions, and expressions, such as `let`, that expand into lambda expressions, is extended to support destructuring of pairs, tuples, lists, and reverse values. Multiple-argument lambda expressions and applications incur implicit structuring and destructuring.

While STALIN $\nabla$  accepts VLAD programs in SCHEME S-expression notation, in this paper, we formulate VLAD programs in a more traditional mathematical notation

<sup>4</sup>VLAD is an acronym for Function Language for AD with a voiced F.

<sup>5</sup>The source code for STALIN $\nabla$  and all of the examples from this paper are available from <http://www-bcl.cs.nuim.ie/~qobi/stalingrad/software/toplas2006.tgz>.

that, *inter alia*, uses infix applications. While  $\text{STALIN}\nabla$  is implemented in SCHEME, not VLAD, in this paper, we use the same mathematical notation both when writing VLAD programs and when specifying the implementation of VLAD. We often have functions in the VLAD language that correspond to functions in the implementation. To avoid confusion in such cases, the language function is referred to as  $t$  while the implementation function is referred to as  $\underline{t}$ .

A preprocessor translates VLAD programs into the pure lambda calculus. Standard SCHEME syntax is expanded using the macros from Kelsey et al. [1998]. Top-level definitions are translated into uses of `letrec`. While  $\text{STALIN}\nabla$  implements `letrec` natively, for expository purposes, in this paper, we assume that `letrec` is implemented in the pure lambda calculus in terms of the Y combinator. Structuring and destructuring is made explicit. Booleans and `ifs` are eliminated and pairs are implemented using the Church encoding:

$$\begin{array}{ll}
 \mathbf{true} & \rightsquigarrow \text{CAR} \\
 \mathbf{false} & \rightsquigarrow \text{CDR} \\
 \mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3 \mathbf{ fi} & \rightsquigarrow (e_1 ((\lambda x e_2), (\lambda x e_3))) [] \text{ where } x \text{ is fresh} \\
 \text{CAR } x & \triangleq x \lambda x_1 \lambda x_2 x_1 \\
 \text{CDR } x & \triangleq x \lambda x_1 \lambda x_2 x_2 \\
 \text{CONS } x_1 x_2 x & \triangleq x x_1 x_2
 \end{array}$$

(The primitives must be aware of the representation of Booleans and pairs.) Finally, quoted constants are replaced with references to variables in the top-level environment.

Given a set  $X$  of *base variables*, a *variable*  $x$  is either a base variable or a *tagged variable*. A tagged variable, in turn, is either a *reverse variable*  $\overleftarrow{x}$ , a *sensitivity variable*  $\overrightarrow{x}$ , or a *backpropagator variable*  $\overline{x}$ . The input program must be formulated using only base variables. The reverse transformation will produce expressions that contain tagged variables. Repeated reverse transformation can yield variables with stacked tags, like  $\overleftarrow{\overrightarrow{\overline{x}}}$ . Variable tagging allows the reverse transformation to generate new variables that do not clash with existing variables and allows a bidirectional correspondence between tagged and untagged variants of a variable.

We assume a total order  $\prec$  on all variables. This will allow unambiguous construction of a list to represent the sensitivity of a function in terms of its free variables.

An *expression*  $e$  is either a *variable access expression*  $x$ , an *application*  $(e_1 e_2)$ , or a *lambda expression*  $(\lambda x e)$ . We often eliminate parenthesis around applications and lambda expressions, taking application to associate to the left and lambda expressions to extend as far right as possible.

Tags on the argument variable  $x$  of a lambda expression allow one to determine whether or not a lambda expression has been transformed, and if so, how many times it has been transformed. We will use this ability, below, to repeatedly untransform a transformed lambda expression to determine the free variables in the original untransformed lambda expression. We also use this ability, below, to construct values that are transformed the same number of times as a correspondingly

transformed lambda expression.

We assume that the bodies of all lambda expressions are converted to A-normal form. An expression in *A-normal form* has the form:

$$\mathbf{let } x_1 \triangleq e_1; \dots; x_m \triangleq e_m \mathbf{ in } x_m \mathbf{ end}$$

where each  $e_i$  is either  $x_j$ ,  $(x_j x_k)$ , or  $(\lambda x e)$ , where  $e$  is in A-normal form. We take **let** to be shorthand for an implementation in the pure lambda calculus in terms of applications and lambda expressions:

$$\begin{aligned} & \mathbf{let } x_1 \triangleq e_1; x_2 \triangleq e_2; \dots; x_m \triangleq e_m \mathbf{ in } e \mathbf{ end} \\ & \sim \mathbf{let } x_1 \triangleq e_1 \mathbf{ in } \mathbf{let } x_2 \triangleq e_2; \dots; x_m \triangleq e_m \mathbf{ in } e \mathbf{ end end} \\ & \mathbf{let } x_1 \triangleq e_1 \mathbf{ in } e \mathbf{ end} \sim ((\lambda x_1 e) e_1) \end{aligned}$$

We further assume that all lambda expressions are  $\alpha$ -converted.

We use  $\mathcal{F} e$  to denote the set of free variables of an expression  $e$ :

$$\begin{aligned} \mathcal{F} x &= \{x\} \\ \mathcal{F} (e_1 e_2) &= (\mathcal{F} e_1) \cup (\mathcal{F} e_2) \\ \mathcal{F} (\lambda x e) &= (\mathcal{F} e) \setminus \{x\} \end{aligned}$$

We use  $\mathcal{B} e$  to denote the set of free variables in the lambda expression  $e$  that correspond to free variables in the original untransformed lambda expression that was (potentially) transformed (multiple times) to yield  $e$ :

$$\begin{aligned} \mathcal{B} (\lambda x e) &= \mathcal{F} (\lambda x e) && \text{when } x \in X \\ \mathcal{B} e &= \{\} && \text{where } \langle \sigma, e \rangle = \overleftarrow{t} \\ \mathcal{B} \overleftarrow{\lambda x e} &= \{\overleftarrow{x'} \mid x' \in \mathcal{B} (\lambda x e)\} \end{aligned}$$

The notation  $\langle \sigma, e \rangle$  and  $\overleftarrow{t}$  used in the second equation above and the notation  $\overleftarrow{\lambda x e}$  used in the third equation above will be defined below. We refer to  $\mathcal{B} e$  as the *base free variables* of  $e$ . The second equation above indicates that we take a lambda expression produced by transforming a primitive  $t$  as having no base free variables.

An *environment*  $\sigma$  is a finite map from variables to values. We use  $\{\}$  to denote the empty environment and use  $\sigma_0$  to denote the top-level environment that contains the standard basis. A *closure* is a pair  $\langle \sigma, e \rangle$ , where  $e$  is a lambda expression and the domain of  $\sigma$  includes  $\mathcal{F} e$ . A *value*  $v$  is either the empty list  $[\ ]$ , a real number  $r$ , a *reverse-tagged value*  $\overleftarrow{v}$  (where  $v$  is  $[\ ]$ , a real, or a reverse-tagged value), a *unary real primitive*  $u$ , a *binary real primitive*  $b$ , a *unary Boolean primitive*  $p$ , a *binary Boolean primitive*  $q$ , an *AD primitive*  $\mathbf{0}$ ,  $\oplus$ ,  $\overleftarrow{\mathcal{J}}$ , or  $\overleftarrow{\mathcal{J}}^{-1}$ , or a closure. Each language primitive  $u$  corresponds to some function  $\underline{u} : \mathbb{R} \rightarrow \mathbb{R}$  in the implementation. Each language primitive  $b$  corresponds to some function  $\underline{b} : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$  in the implementation. Each language primitive  $p$  corresponds to some unary predicate  $\underline{p}$ , over values, in the implementation. Each language primitive  $q$  corresponds to some binary relation  $\underline{q}$ , over pairs of values, in the implementation. The language primitives  $\mathbf{0}$ ,  $\oplus$ ,  $\overleftarrow{\mathcal{J}}$ , and  $\overleftarrow{\mathcal{J}}^{-1}$  correspond to the functions  $\underline{\mathbf{0}}$ ,  $\underline{\oplus}$ ,  $\underline{\overleftarrow{\mathcal{J}}}$ , and  $\underline{\overleftarrow{\mathcal{J}}^{-1}}$  in

the implementation. We use  $t$  to denote any primitive and refer to closures and primitives collectively as *functions*.

Recall that tagged argument variables of lambda expressions indicate that those lambda expressions have been transformed. This allows one to determine whether a closure has been transformed. Value tags are used to tag non-closure values as having been transformed. Value tags stack to indicate multiply-transformed values, much like variable tags stack. In the absence of sensitivity and backpropagator tags, correct programs obey two simple invariants. First, the tag stack of a function must be a prefix of the tag stack of an argument to which the function is applied. Second, the tag stack of a variable must be a prefix of the tag stack of a value to which the variable is bound.

We use  $(v_1, v_2)$  as shorthand for the *Church pair*:

$$\langle \{(x_1 \mapsto v_1), (x_2 \mapsto v_2)\}, (\lambda x_3 x_3 x_1 x_2) \rangle$$

We often eliminate parentheses around pairs, taking comma to associate to the right. We use  $[v_1, \dots, v_n]$  to denote a list. It is shorthand for  $(v_1, \dots, v_n, [])$ . To maintain the above invariants, we need to introduce transformed pairs and lists. We use  $(v_{1,x} v_2)$ ,  $[]_x$ , and  $[v_1, \dots, v_n]_x$  to denote pairs, empty lists, and lists that have been transformed according to the tag stack on the variable  $x$ :

$$\begin{aligned} (v_{1,x} v_2) &= (v_1, v_2) && \text{when } x \in X \\ ((\overleftarrow{\mathcal{J}} v_1), \overleftarrow{x} (\overleftarrow{\mathcal{J}} v_2)) &= \overleftarrow{\mathcal{J}} (v_{1,x} v_2) \\ []_x &= [] && \text{when } x \in X \\ []_{\overleftarrow{x}} &= \overleftarrow{[]}_x \\ [v_1, \dots, v_n]_x &= (v_{1,x} \dots_{,x} v_{n,x} []_x) \end{aligned}$$

The implementation  $\overleftarrow{\mathcal{J}}$  of the primitive  $\overleftarrow{\mathcal{J}}$  will be defined below. Note that pair and list formation, both in the base case, as well as in the transformed case, is invertible. Thus we often use this notation to indicate destructuring. Also note that we use this notation both when we write VLAD expressions and when we specify functions that implement VLAD.

We define the notion of *conformance* between two values as follows. The empty list conforms to itself. Two reals are conformant. A reverse-tagged value  $\overleftarrow{v}_1$  conforms to another reverse-tagged value  $\overleftarrow{v}_2$  if  $v_1$  conforms to  $v_2$ . A primitive conforms to itself. Two environments are conformant if they have the same domains and both environments map each given variable to conformant values. Two closures are conformant if their environments are conformant and they have equivalent expressions.

We define an addition operation  $\underline{\oplus}$  between two conformant values as follows:

$$\begin{aligned}
[] \underline{\oplus} [] &= [] \\
r_1 \underline{\oplus} r_2 &= r_1 + r_2 \\
\overleftarrow{v}_1 \underline{\oplus} \overleftarrow{v}_2 &= \overleftarrow{v_1 \underline{\oplus} v_2} \quad \text{where } \overleftarrow{v}_1 \text{ and } \overleftarrow{v}_2 \text{ are reverse-tagged values} \\
t \underline{\oplus} t &= t \\
(\sigma_1 \underline{\oplus} \sigma_2) x &= (\sigma_1 x) \underline{\oplus} (\sigma_2 x) \\
\langle \sigma_1, e \rangle \underline{\oplus} \langle \sigma_2, e \rangle &= \langle (\sigma_1 \underline{\oplus} \sigma_2), e \rangle
\end{aligned}$$

We define the notion of *match* between a value and a corresponding sensitivity as follows. The empty list matches itself. Two reals match. A reverse-tagged value  $\overleftarrow{v}_1$  matches another reverse-tagged value  $\overleftarrow{v}_2$  if  $v_1$  matches  $v_2$ . A primitive matches the empty list. A closure  $\langle \sigma, (\lambda x e) \rangle$  matches a list  $[v_1, \dots, v_n]_x$  when  $x_1, \dots, x_n$  are the elements of  $\mathcal{B}(\lambda x e)$  ordered by  $\prec$  and each  $\sigma x_i$  matches  $v_i$ .

A *zero* is either  $[]$ ,  $0$ , or a closure whose environment maps every variable to a (possibly different) zero. Every value has exactly one matching zero. We use  $\underline{\mathbf{0}}$   $v$  to denote the zero that matches  $v$ :

$$\begin{aligned}
\underline{\mathbf{0}} [] &= [] \\
\underline{\mathbf{0}} r &= 0 \\
\underline{\mathbf{0}} \overleftarrow{v} &= \underline{\mathbf{0}} \overleftarrow{v} \quad \text{where } \overleftarrow{v} \text{ is a reverse-tagged value} \\
\underline{\mathbf{0}} t &= [] \\
\underline{\mathbf{0}} \langle \sigma, (\lambda x e) \rangle &= [(\underline{\mathbf{0}}(\sigma x_1)), \dots, (\underline{\mathbf{0}}(\sigma x_n))]_x \\
&\quad \text{where } x_1, \dots, x_n \text{ are the elements of } \\
&\quad \mathcal{B}(\lambda x e) \text{ ordered by } \prec
\end{aligned}$$

To define the reverse transform, we first define the following transformations on **let** bindings:

$$\begin{aligned}
\phi\{x_i \triangleq x_j\} &= \overleftarrow{x_i} \triangleq \overleftarrow{x_j} \\
\phi\{x_i \triangleq x_j x_k\} &= (\overleftarrow{x_j}, \overleftarrow{x_i}) \triangleq \overleftarrow{x_j} \overleftarrow{x_k} \\
\phi\{x_i \triangleq \lambda x e\} &= \overleftarrow{x_i} \triangleq \overleftarrow{\lambda x e} \\
\rho\{x_i \triangleq x_j\} &= \overleftarrow{x_j} \oplus \triangleq \overleftarrow{x_i} \\
\rho\{x_i \triangleq x_j x_k\} &= (\overleftarrow{x_j}, \overleftarrow{x_k}) \oplus \triangleq \overleftarrow{x_j} \overleftarrow{x_k} \\
\rho\{x_i \triangleq \lambda x e\} &= [\overleftarrow{x'_1}, \dots, \overleftarrow{x'_l}]_x \oplus \triangleq \overleftarrow{x_i} \quad \text{where } x'_1, \dots, x'_l \text{ are the elements of } \\
&\quad \mathcal{B}(\lambda x e) \text{ ordered by } \prec
\end{aligned}$$

We use  $\phi$  to denote the forward-phase transformation and  $\rho$  to denote the reverse-



that it maps  $x$  to  $v$ . We have the following standard ‘eval/apply’ evaluator:

$$\begin{aligned}
\mathcal{E} \sigma x &= \sigma x \\
\mathcal{E} \sigma (e_1 e_2) &= \mathcal{A} (\mathcal{E} \sigma e_1) (\mathcal{E} \sigma e_2) \\
\mathcal{E} \sigma (\lambda x e) &= \langle \sigma, (\lambda x e) \rangle \\
\\
\mathcal{A} u v &= \underline{u} v \\
\mathcal{A} \underline{b} (v_1, v_2) &= \underline{b} v_1 v_2 \\
\mathcal{A} \underline{p} v &= \langle \{\}, (\lambda x_1 x_1 \lambda x_2 \lambda x_3 x_2) \rangle \text{ when } \underline{p} v \\
\mathcal{A} \underline{p} v &= \langle \{\}, (\lambda x_1 x_1 \lambda x_2 \lambda x_3 x_3) \rangle \text{ when } \neg(\underline{p} v) \\
\mathcal{A} \underline{q} (v_1, v_2) &= \langle \{\}, (\lambda x_1 x_1 \lambda x_2 \lambda x_3 x_2) \rangle \text{ when } \underline{q} v_1 v_2 \\
\mathcal{A} \underline{q} (v_1, v_2) &= \langle \{\}, (\lambda x_1 x_1 \lambda x_2 \lambda x_3 x_3) \rangle \text{ when } \neg(\underline{q} v_1 v_2) \\
\mathcal{A} \underline{\mathbf{0}} v &= \underline{\mathbf{0}} v \\
\mathcal{A} \underline{\oplus} (v_1, v_2) &= v_1 \underline{\oplus} v_2 \\
\mathcal{A} \overleftarrow{\mathcal{J}} v &= \overleftarrow{\mathcal{J}} v \\
\mathcal{A} \overleftarrow{\mathcal{J}}^{-1} v &= \overleftarrow{\mathcal{J}}^{-1} v \\
\mathcal{A} \langle \sigma, (\lambda x e) \rangle v &= \mathcal{E} \sigma [x \mapsto v] e
\end{aligned}$$

All that remains is to show how to transform the primitives  $t$  into  $\overleftarrow{t}$ . We first do that for the primitives  $u$ ,  $b$ ,  $p$ , and  $q$  as follows:

$$\begin{aligned}
\overleftarrow{u} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} x) ((\overleftarrow{\mathcal{J}} (u x)), (\lambda \overleftarrow{y} ([], ((\mathcal{D} \underline{u} x) \times \overleftarrow{y})))) \\
\overleftarrow{b} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} (x_1, x_2)) \\
&\quad ((\overleftarrow{\mathcal{J}} (b (x_1, x_2))), \\
&\quad (\lambda \overleftarrow{y} ([], (((\mathcal{D}_1 \underline{b} (x_1, x_2)) \times \overleftarrow{y}), ((\mathcal{D}_2 \underline{b} (x_1, x_2)) \times \overleftarrow{y})))))) \\
\overleftarrow{p} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} x) ((\overleftarrow{\mathcal{J}} (p x)), (\lambda \overleftarrow{y} ([], (\mathbf{0} x)))) \\
\overleftarrow{q} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} (x_1, x_2)) ((\overleftarrow{\mathcal{J}} (q (x_1, x_2))), (\lambda \overleftarrow{y} ([], ((\mathbf{0} x_1), (\mathbf{0} x_2))))))
\end{aligned}$$

In the above,  $\mathcal{D} \underline{u} x$  denotes a VLAD expression that evaluates the derivative of  $\underline{u}$  at  $x$  and  $\mathcal{D}_1 \underline{b} (x_1, x_2)$  and  $\mathcal{D}_2 \underline{b} (x_1, x_2)$  denote VLAD expressions that evaluate the partial derivatives of  $\underline{b}$ , with respect to its first and second arguments, at  $(x_1, x_2)$ . Note that since  $\overleftarrow{t}$  denotes a (transformed) *value*, we generate such a value, i.e. a closure, by evaluating a lambda expression in the top-level environment.

Closure now requires transformations of the AD primitives:

$$\begin{aligned}
\overleftarrow{\mathbf{0}} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} x) ((\overleftarrow{\mathcal{J}} (\mathbf{0} x)), (\lambda \overleftarrow{y} ([], (\mathbf{0} x)))) \\
\overleftarrow{\oplus} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} (x_1, x_2)) ((\overleftarrow{\mathcal{J}} (x_1 \oplus x_2)), (\lambda \overleftarrow{y} ([], (\overleftarrow{y}, \overleftarrow{y})))) \\
\overleftarrow{\mathcal{J}} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} x) ((\overleftarrow{\mathcal{J}} (\overleftarrow{\mathcal{J}} x)), (\lambda \overleftarrow{y} ([], (\overleftarrow{\mathcal{J}}^{-1} \overleftarrow{y})))) \\
\overleftarrow{\mathcal{J}}^{-1} &= \mathcal{E} \sigma_0 \lambda (\overleftarrow{\mathcal{J}} x) ((\overleftarrow{\mathcal{J}} (\overleftarrow{\mathcal{J}}^{-1} x)), (\lambda \overleftarrow{y} ([], (\overleftarrow{\mathcal{J}} \overleftarrow{y}))))
\end{aligned}$$

Two subtle issues remain to be addressed. First, Boolean primitives  $p$  and  $q$  might take constant time even when given input of arbitrary size. Examples include the SCHEME type predicates like REAL?. Transformations of these predicates, however, involve applying the inverse transformation  $\overleftarrow{\mathcal{J}}^{-1}$  to their input. The implementation  $\overleftarrow{\mathcal{J}}^{-1}$  nominally traverses its input. The commensurate increase in temporal

complexity can be avoided by taking the implementations  $\underline{\mathbf{Q}}$ ,  $\underline{\oplus}$ ,  $\overleftarrow{\mathcal{J}}$ , and  $\overleftarrow{\mathcal{J}}^{-1}$  of the AD primitives to be lazy.

Second, standard implementations of languages like SCHEME allow structure sharing. This allows code like:

```

let  $x_1 = (0, 0)$ ;
       $x_2 = (x_1, x_1)$ ;
       $\vdots$ 
       $x_n = (x_{n-1}, x_{n-1})$ 
in  $x_n$  end

```

to produce a linear-sized representation of an exponentially-sized value in linear time. AD primitives, like  $\overleftarrow{\mathcal{J}}$ , that traverse their input can nominally yield output whose size is exponential in the size of the input. This can be avoided by memoizing the implementations  $\underline{\mathbf{Q}}$ ,  $\underline{\oplus}$ ,  $\overleftarrow{\mathcal{J}}$ , and  $\overleftarrow{\mathcal{J}}^{-1}$  of the AD primitives, when applied to non-scalar input, to preserve the structure sharing.

The above two cases were the only opportunities in the transformed code for the temporal complexity to exceed that of the primal computation by more than a small constant factor, as can be verified by a tedious case analysis of every transformation rule above. For this reason, with the above provisos concerning memoization and lazy computation of the AD primitives, if we let  $(y, \overline{y}) = \overleftarrow{\mathcal{J}} f x$  then the number of primitive arithmetic operations performed while evaluating  $\overleftarrow{\mathcal{J}} f x$  is the same as when evaluating  $f x$ , and the number of primitive operations performed while evaluating  $\overline{x} = \overline{y} \overline{y}$  is also the same, up to a small constant factor. This was confirmed for a small suite of benchmark problems, included in the distribution, by instrumenting the STALIN $\nabla$  implementation to count primitive arithmetic operations.

## 5. EXAMPLES OF THE UTILITY OF OUR METHOD

As mentioned in the introduction, to achieve closure, our method solves two technical problems:

- It supports transformation of nested lambda expressions, particularly those with free-variable references. Moreover, it can handle the case where reverse-mode AD is applied to a function  $f$  that takes an argument  $x$  and that, in turn, applies reverse-mode AD to a function  $g$ , nested inside  $f$ , that has a free reference to  $x$ , i.e. the argument to the surrounding function  $f$ .
- It supports application of  $\overleftarrow{\mathcal{J}}$  to itself.

We present two different examples, both of which illustrate the utility of our solution to these two technical problems. These examples run in our prototype implementation and are included in the distribution. We know of no other approach to reverse-mode AD that can handle these examples. Furthermore, our distribution contains benchmark scripts that use the metering facility of our prototype implementation to illustrate that our approach has the correct temporal complexity properties.

As a first example, consider a continuous two-person zero-sum game. Unlike conventional discrete games, where the two players select from finite sets of  $m$  and  $n$

strategies and the payoff is specified by an  $m \times n$  matrix, our players select from multidimensional continuous strategies in  $\mathbb{R}^m$  and  $\mathbb{R}^n$  and the payoff is specified by a function  $\mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$ . We wish to find the optimal minimax strategy, i.e. a saddle point in the payoff function. In traditional notation, this corresponds to computing  $\min_x \max_y f(x, y)$ .

The gradient of a function can be computed with:

$$\nabla f x \triangleq \text{CDR} ((\text{CDR} ((\overleftarrow{\mathcal{J}} f) (\overleftarrow{\mathcal{J}} x))) 1)$$

We then construct `UNIVARIATEARGMIN` ( $f, \epsilon$ ), a univariate minimizer based on the golden-section algorithm using a translation of the `mnbrak` and `golden` functions from Press et al. [1992] into `VLAD`. We omit the translation for brevity. We then construct a multivariate minimizer based on gradient descent:

```

ARGMIN ( $f, x_0, \epsilon$ )  $\triangleq$ 
  let  $g \triangleq \nabla f x_0$ 
  in if  $\|g\| \leq \epsilon$ 
    then  $x$ 
    else ARGMIN
      ( $f,$ 
       ( $x_0 + ((\text{UNIVARIATEARGMIN} ((\lambda k f (x_0 + (k \times g))), \epsilon)) \times g)$ ),
        $\epsilon$ ) fi end

```

using the univariate minimizer to perform line search. From this, we can construct:

$$\begin{aligned} \text{ARGMAX} (f, x_0, \epsilon) &\triangleq \text{ARGMIN} ((\lambda x - (f x)), x_0, \epsilon) \\ \text{MAX} (f, x_0, \epsilon) &\triangleq f (\text{ARGMAX} (f, x_0, \epsilon)) \end{aligned}$$

Now let us construct a simple payoff function:

$$\text{PAYOFF} ([s, t], [u, v]) \triangleq s^2 + t^2 - u^2 - v^2$$

The optimal strategy  $(x^*, y^*) = ([0, 0], [0, 0])$  can be found using:

```

let  $x^* \triangleq \text{ARGMIN} ((\lambda x \text{MAX} ((\lambda y \text{PAYOFF} (x, y)), y_0, \epsilon)), x_0, \epsilon)$ 
in ( $x^*, (\text{ARGMAX} ((\lambda y \text{PAYOFF} (x^*, y)), y_0, \epsilon))$ ) end

```

Note that finding  $x^*$  involves taking the derivative of  $\lambda x \dots$  which in turn, involves taking the second derivative of  $\lambda y \dots$ . Also note that  $\lambda y \dots$  has a free reference to  $x$ , which is the argument to  $\lambda x \dots$ . Finally note that  $\lambda x \dots$  calls `MAX`, which calls `ARGMAX`, which calls `ARGMIN`, which calls  `$\nabla$` , which calls  `$\overleftarrow{\mathcal{J}}$` . Since finding  $x^*$  involves taking the derivative of  $\lambda x \dots$ , this ultimately involves applying  `$\overleftarrow{\mathcal{J}}$`  to itself.

As a second example, consider the computation and use of Hessians. A common misconception is that numerical methods based on Hessians are inefficient. While it is true that explicitly storing a Hessian matrix takes space that is quadratic in the input size, and thus explicitly computing a Hessian matrix takes time that is at least quadratic in the input size, one can compute the product of a Hessian matrix and an arbitrary vector, with the same temporal complexity (up to a small

constant factor) as computing the original function [Christianson, 1992; Werbos, 1992; Pearlmutter, 1994]. We do so now using double application of reverse-mode AD. Let  $\mathcal{H}$  denote a higher-order function that maps a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  to a function of type  $\mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$  that computes the Hessian matrix of  $f$  at a real vector. The quantity  $(\mathcal{H} f \mathbf{x}) \times \mathbf{v}$  can be computed as:

$$(\text{CDR} ((\overleftarrow{\mathcal{J}} (\lambda x (\text{CDR} ((\overleftarrow{\mathcal{J}} f) (\overleftarrow{\mathcal{J}} x))) 1)) \cdot \mathbf{v})) (\overleftarrow{\mathcal{J}} x))) 1))$$

where  $\cdot$  denotes vector dot product. If we take  $f [x, y] \triangleq 2x^2 + 3xy + 4y^2$  then  $(\mathcal{H} f [3, 4]) \times [7, 8] = [52, 85]$ .

Note that computing the Hessian obviously involves taking second derivatives, which in our case involves transforming the result of a transformation. Also note that  $\lambda x \dots$  has a free reference to  $\mathbf{v}$ . Finally note that the outer  $\overleftarrow{\mathcal{J}}$  transforms  $\lambda x \dots$ , which calls  $\overleftarrow{\mathcal{J}}$ . This ultimately involves applying  $\overleftarrow{\mathcal{J}}$  to itself.

## 6. PRIOR WORK

The numeric part of the primal computation can be thought of as a data-flow graph leading from input real numbers to output real numbers. We can concentrate on this data-flow graph, and ignore all other parts of the computation as mere scaffolding. In this context, reverse-mode AD refers to a particular construction in which the primal data-flow graph is transformed to construct an adjoint graph that computes the sensitivity values. In the adjoint, the direction of the data-flow edges are reversed; addition nodes are replaced by fanout nodes; fanout nodes are replaced by addition nodes; and other nodes are replaced by multiplication by their linearizations. The main constructions of this paper can, in this context, be viewed as a method for constructing scaffolding that supports this adjoint computation.

Karczmarszuk [2000a,b, 2001a] presents a method that we will refer to as KM, and claims that KM constitutes an implementation of reverse-mode AD in HASKELL. In the absence of nesting, KM does calculate correct gradients in HASKELL. However KM is not actually reverse-mode AD; rather, as we show below, it is an extremely inefficient implementation of forward-mode AD. KM actually calculates gradients by pairing each primal value with a vector of  $n$  perturbation values, taking the perturbations of the inputs to be the columns of an identity matrix. This imposes an overhead of  $O(n)$  in both time and space as compared with the primal computation, while a correctly implemented reverse-mode would require only  $O(1)$  overhead in time. However, KM also uses a highly inefficient lazy representation of real numbers for perturbations. This representation multiplies out the adjoint graph to distribute multiplication over addition, thus converting the graph into an exponentially larger sum of products. This results in an additional exponential overhead in both time and space.

More specifically, KM represents perturbations  $\overline{x}$  as closures  $(\lambda z \overline{x} \times z)$  and implements arithmetic on these in a very inefficient fashion. We will refer to such values as  $\tilde{\mathbb{R}}$  numbers. Since  $\tilde{\mathbb{R}}$  numbers are used only to represent perturbations, only two arithmetic primitives need be defined: addition of two  $\tilde{\mathbb{R}}$  numbers, and multiplication of a conventional  $\mathbb{R}$  by an  $\tilde{\mathbb{R}}$  number. In KM, these are implemented

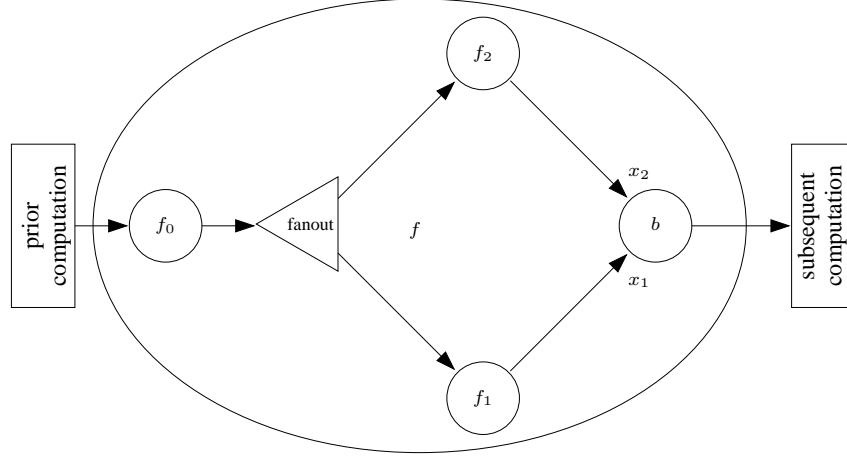


Fig. 1. With our method, the backpropagator for  $f$  calls the backpropagators for  $f_1$  and  $f_2$ , sums their result at the fanout point, and then calls the backpropagator for  $f_0$  and the prior computation only once. Fanout in the primal computation causing addition in the reverse computation is the key insight of reverse-mode AD.

as:

$$\begin{aligned} (x : \mathbb{R}) \tilde{\times} (y : \tilde{\mathbb{R}}) &\triangleq \lambda z y (x \times z) \\ (x : \tilde{\mathbb{R}}) \tilde{+} (y : \tilde{\mathbb{R}}) &\triangleq \lambda z (x z) + (y z) \end{aligned}$$

which defers all arithmetic until an  $\tilde{\mathbb{R}}$  number is converted to a conventional  $\mathbb{R}$  by applying it to the number 1, at the expense of a potentially exponential amount of unnecessary computation.

In KM, arithmetic is then overloaded to carry perturbations, represented as  $\tilde{\mathbb{R}}$  numbers, along with the primals. Using the notation of Section 2:

$$\begin{aligned} \overrightarrow{u} (x, \overrightarrow{\mathbf{x}}) &= ((u x), (\text{MAP } (\lambda \overrightarrow{x'} (\mathcal{D} u x) \tilde{\times} \overrightarrow{x'}) \overrightarrow{\mathbf{x}})) \\ \overrightarrow{b} ((x_1, \overrightarrow{\mathbf{x}}_1), (x_2, \overrightarrow{\mathbf{x}}_2)) &= ((b (x_1, x_2)), \\ &\quad (\text{MAP2 } (\lambda (\overrightarrow{x}_1, \overrightarrow{x}_2) ((\mathcal{D}_1 b (x_1, x_2)) \tilde{\times} \overrightarrow{x}_1) \tilde{+} \\ &\quad \quad \quad ((\mathcal{D}_2 b (x_1, x_2)) \tilde{\times} \overrightarrow{x}_2)) \\ &\quad \quad (\overrightarrow{\mathbf{x}}_1, \overrightarrow{\mathbf{x}}_2))) \end{aligned}$$

Note that multiplication of an  $\tilde{\mathbb{R}}$  number by a conventional  $\mathbb{R}$  defers the computation, and requires unnecessary space, but does not require unnecessary primitive multiplication operations. Rather, it is fanout of  $\tilde{\mathbb{R}}$  values that causes the same  $\tilde{\mathbb{R}}$  number to be called repeatedly with different arguments, via the addition operation for  $\tilde{\mathbb{R}}$  numbers. Because an  $\tilde{\mathbb{R}}$  number which has undergone fanout would in general be called repeatedly with different arguments, common subexpression elimination or memoization would not prevent the exponential inefficiency of  $\tilde{\mathbb{R}}$  arithmetic.

## 7. DISCUSSION

The primary technical difficulty solved above is proper conversion of fanout in the primal to addition in the adjoint graph, in the face of closures and environments. The required bookkeeping is shown diagrammatically in Figure 1. If a program was constructed solely out of unary functions, it could not have fanout and thus would not be subject to these issues. Thus it may seem paradoxical, at first, that so much bookkeeping is needed in VLAD to handle fanout, since like ML, VLAD supports only unary functions and unary primitives. The paradox is resolved by noticing that function application itself is a binary function! This is manifest in our method for supporting free variables: having backpropagators return an environment sensitivity paired with an input sensitivity and accumulating the environment sensitivity into the sensitivity of the target of an application and the input sensitivity into the sensitivity of the argument of that application. VLAD, like ML, uses tupling and currying to implement functions that take multiple arguments. With Church pairs, tupling, in turn, reduces to currying, which in turn, requires free variables.

It is also interesting to note that we have not eliminated the ‘tape’ from reverse-mode AD. That would be impossible, because the tape stores intermediate values computed during the forward phase that are needed during the reverse phase. What we have done is to change the representation of the tape from an interpreted (or runtime compiled) data structure to pre-compiled closures. The traditional tape stores not only values but also operations on those values. The dichotomy between storing values and operations is reflected in our method by the fact that closures have environments to store values and expressions to store operations. Herein lies the difference: multiple closures with different environments can share the same expression. Using closures to represent the tape allows factoring out common sequences of operations performed on different values. This representation also exposes the tape to the compiler and to other general-purpose mechanisms, including the  $\overleftarrow{\mathcal{J}}$  operator itself.

## 8. CONCLUSION

We have shown a novel method for implementing reverse-mode AD in a functional framework. Our method exhibits three important closure properties:

- (1) It applies to any lambda-calculus expression, including those with free variables.
- (2) The transformation of a lambda-calculus expression is itself a lambda-calculus expression allowing repeated application to compute higher-order derivatives.
- (3) The temporal complexity of a function is preserved under transformation.

Traditional implementations of reverse mode exhibits 3 but not 1 and 2.

Our method involves a global program transformation, implemented by a novel first-class programming-language primitive  $\overleftarrow{\mathcal{J}}$ , rather than a local transformation, implemented by overloading. This allows application of the reverse-mode transformation by programs within the language, rather than by a preprocessor. To achieve closure, we solved two technical problems: supporting transformation of nested lambda expressions with free-variable references, and application of  $\overleftarrow{\mathcal{J}}$  to itself. We have illustrated the utility of the solution to these two technical prob-

lems with two practical examples, namely finding saddle points and computing Hessian-vector products.

## REFERENCES

- APPEL, A. W. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (Apr.), 17–20.
- CHRISTIANSON, B. 1992. Automatic Hessians by reverse accumulation. *IMA J. of Numerical Analysis* 12, 135–150.
- CORLISS, G., FAURE, C., GRIEWANK, A., HASCOËT, L., AND NAUMANN, U. 2001. *Automatic Differentiation: From Simulation to Optimization*. Springer-Verlag, New York, NY.
- GRIEWANK, A. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM.
- KARCZMARCZUK, J. 1998a. Functional differentiation of computer programs. In *Proceedings of the III ACM SIGPLAN International Conference on Functional Programming*. Baltimore, MD, 195–203.
- KARCZMARCZUK, J. 1998b. Lazy differential algebra and its applications. In *Workshop, III International Summer School on Advanced Functional Programming*. Braga, Portugal.
- KARCZMARCZUK, J. 1999. Functional coding of differential forms. In *Scottish Workshop on FP*.
- KARCZMARCZUK, J. 2000a. Adjoint codes in functional framework. <http://users.info.unicaen.fr/~karczma/arpap/revdiff.ps>.
- KARCZMARCZUK, J. 2000b. Lazy time reversal, and automatic differentiation. <http://users.info.unicaen.fr/~karczma/arpap/revpearl.pdf>.
- KARCZMARCZUK, J. 2001a. Calcul des adjoints et programmation paresseuse. <http://users.info.unicaen.fr/~karczma/arpap/jflarev.pdf>.
- KARCZMARCZUK, J. 2001b. Functional differentiation of computer programs. *Journal of Higher-Order and Symbolic Computation* 14, 35–57.
- KEDEM, G. 1980. Automatic differentiation of computer programs. *ACM Trans. on Mathematical Software* 6, 2, 150–65.
- KELSEY, R., CLINGER, W., AND REES, J. 1998. Revised<sup>5</sup> report on the algorithmic language SCHEME. *Higher-Order and Symbolic Computation* 11, 1 (Sept.).
- KELSEY, R. A. 1995. 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 (Mar.), 13–22.
- MONAGAN, M. B. AND NEUENSCHWANDER, W. M. 1993. GRADIENT: Algorithmic differentiation in Maple. In *International Symposium on Symbolic and Algebraic Computation*.
- PEARLMUTTER, B. A. 1994. Fast exact multiplication by the Hessian. *Neural Computation* 6, 1, 147–160.
- PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C*, second ed. Cambridge University Press, New York, NY.

- RALL, L. B. 1981. *Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science 120*. Springer-Verlag, New York, NY.
- RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. 1986. Learning representations by back-propagating errors. *Nature* 323, 533–6.
- SABRY, A. AND FELLEISEN, M. 1993. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* 6, 3–4, 289–360.
- SISKIND, J. M. 1999. Flow-directed lightweight closure conversion. Technical report 99–190R, NEC Research Institute, Inc.
- SPEELPENNING, B. 1980. Compiling fast partial derivatives of functions given by algorithms. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.
- SUSSMAN, G. J., WISDOM, J., AND MAYER, M. E. 2001. *Structure and Interpretation of Classical Mechanics*. MIT Press, Cambridge, MA.
- WENGERT, R. E. 1964. A simple automatic derivative evaluation program. *Communications of the ACM* 7, 8, 463–4.
- WERBOS, P. J. 1992. Neural networks, system identification, and control in the chemical process industries. In *Handbook of Intelligent Control—Neural, Fuzzy, and Adaptive approaches*, D. A. White and D. A. Sofge, Eds. Van Norstrand Reinhold, Chapter 10, 283–356. see section 10.7.

Received Month Year; revised Month Year; accepted Month Year