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

BARAK A. PEARLMUTTER
Hamilton Institute
and
JEFFREY MARK SISKIND
Purdue University

We show that reverse-mode AD (Automatic Differentiation)—a generalized gradient-calculation operator—can be incorporated as a first-class function in an augmented lambda calculus, and therefore into a functional-programming language. Closure is achieved, in that the new operator can be applied to any expression in the augmented language, yielding an expression in that language. This requires the resolution of two major technical issues: (a) how to transform nested lambda expressions, including those with free-variable references, and (b) how to support self application of the AD machinery. AD transformations preserve certain complexity properties, among them that the reverse phase of the reverse-mode AD transformation of a function have the same temporal complexity as the original untransformed function. First-class unrestricted AD operators increase the expressive power available to the numeric programmer, and may have significant practical implications for the construction of numeric software that is robust, modular, concise, correct, and efficient.

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, derivatives, forward-mode AD, higher-order AD, higher-order functional languages, Jacobian, program transformation, reflection

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

Pearlmutter was supported, in part, by Science Foundation Ireland grant `00/PI.1/C067` and the Higher Education Authority of Ireland. Siskind was supported, in part, by NSF grant CCF-0438806. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.
Authors' addresses: B. A. Pearlmutter, Hamilton Institute and Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland; email: `barak@cs.nuim.ie`; J. M. Siskind (contact author), School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, Room 330, West Lafayette, IN 47907-2035 USA; email: `qobi@purdue.edu`.
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

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

This traditional method for taking derivatives, however, has an 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 derivative of a product of n factors:

$$(u_1 u_2 \cdots u_n)' = \underbrace{(u_1' u2 \cdots u_n) + (u_1 u_2' \cdots u_n) + \cdots + (u_1 u_2 \cdots u_{n-1} u_n')}_{n \text{ terms}}$$

has n terms, each of which has n factors, and thus has length that is not linear in n. In general, this length cannot be substantially reduced without the introduction of temporaries. Evaluating derivatives could take dramatically more time than evaluating the original expressions. While this may be of little consequence in classical 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 constant factor. Moreover, the temporal complexity of evaluating a transformed expression is the same as that of the original expression.

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 by transforming the computation graph rather than by 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.[1]

Computing the gradient of a function $\mathbb{R}^n \rightarrow \mathbb{R}$ using forward-mode AD requires n applications of forward-mode AD, imposing an $O(n)$ factor slowdown. The same

[1]Forward-mode AD has been previously applied to functional programs, as discussed in Section 2. 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 exception of the tutorial in Section 2, this paper focuses solely on reverse mode.

gradient can also be computed using a single application of reverse-mode AD, which would impose only a constant factor slowdown. For this reason, the methods we describe may enjoy significant practical application in 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.

Traditional implementations of reverse-mode AD often lack the closure property. Derivatives are typically 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 *non-local* program transformation. This allows application of the reverse-mode AD 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 addresses two technical issues. 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.[2]

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 to building better functional-programming implementations for scientific computing,

[2]An attempt to create a typed lambda calculus incorporating $\overleftarrow{\mathcal{J}}$ must address the issue of giving $\overleftarrow{\mathcal{J}}$ a polymorphic type while allowing self-application like $(\overleftarrow{\mathcal{J}}\ \overleftarrow{\mathcal{J}})$. Such self-application cannot be prohibited because it arises internally from any nested application of AD.

but also to building more powerful AD systems for conventional languages. For example, many 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 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 prior related work on reverse-mode AD. Section 7 discusses fanout and the relationship between fanout, binary functions, and free variables. Section 8 summarizes 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.

For much of this paper, we use x to denote real scalars, $\mathbf{x}$ to denote real (column) vectors, $\mathbf{X}$ to denote real matrices, u to denote functions from real scalars to real scalars, b to denote functions from pairs of real scalars to real scalars, f to denote functions from real vectors to real vectors or from real vectors to real scalars, and juxtaposition to denote function application, which we take to associate to the left. We use subscripts to distinguish variables, and indicate indexing of vectors and matrices with square brackets. We use comma to indicate pair and tuple formation, [ ] to denote the empty list, and square brackets to also denote list formation. Whether square brackets denote indexing or list formation will be clear from context. Infix $\circ$ denotes function composition, $+$ denotes either scalar or vector addition, and $\times$ denotes multiplication: either a matrix by a matrix, a matrix by a vector, a scalar by a vector, or a scalar by a scalar. Multiplication has higher precedence than addition. A superscript $\top$ denotes matrix transposition, so $(\mathbf{X}_1 \times \mathbf{X}_2)^\top = \mathbf{X}_2^\top \times \mathbf{X}_1^\top$.

$\mathcal{D}$ denotes the higher-order function that maps functions u to functions that compute the derivative of u, and $\mathcal{D}_1$ and $\mathcal{D}_2$ 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. ∇ and $\mathcal{J}$ denote the higher-order functions that map functions f to functions that compute the gradient vector or the Jacobian matrix, respectively, of f, at a real vector.[3] We use $=$ for equations,

[3] In classical differential calculus, the gradient of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ at $\mathbf{x}$ is defined as $\nabla f(\mathbf{x}) = \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right)$. The Jacobian generalizes the notion of the gradient to functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. The $m \times n$ Jacobian matrix $\mathbf{J}$ of f at $\mathbf{x}$ has entries $J_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$. Matrices can be viewed both as data and as linear functions. The functional view comes from the fact that one can multiply a matrix by a vector to yield a vector. In this sense, we can view the above Jacobian matrix as a (linear) function $\mathbf{J} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and further view its transpose as a (linear) function

$:=$ for assignments, $\equiv$ for mathematical definitions and meta-level definitions used outside a program, $\stackrel{\triangle}{=}$ for definitions and **let** bindings within a program, and $\leadsto$ for program transformations. We use semicolon to indicate sequencing of assignments and **let** bindings (i.e., as in `let*` in SCHEME). We further use $x \mathrel{+{:=}} e$ as shorthand for $x := x + e$, and similarly for $\oplus{:=}$ and $\oplus\stackrel{\triangle}{=}$. (The function $\oplus$ will be defined later.) We adopt the standard convention that the scope of lambda binders extends as far right as possible. Because all lambda expressions used in this paper bind a single variable, we omit the conventional dot between the bound variable and the body.

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 in $\mathbb{R}^m$, $\mathbf{x}_0$ 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 $\mathbf{x}_0$:

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

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

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

which computes $\overleftharpoon{\mathbf{X}_0} = (\mathcal{J}\ f\ \mathbf{x}_0)^\top$. These have a downside: storage of the intermediate $\overrightharpoon{\mathbf{X}_i}$ and $\overleftharpoon{\mathbf{X}_i}$ 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

$\mathbf{J}^\top : \mathbb{R}^m \to \mathbb{R}^n$.

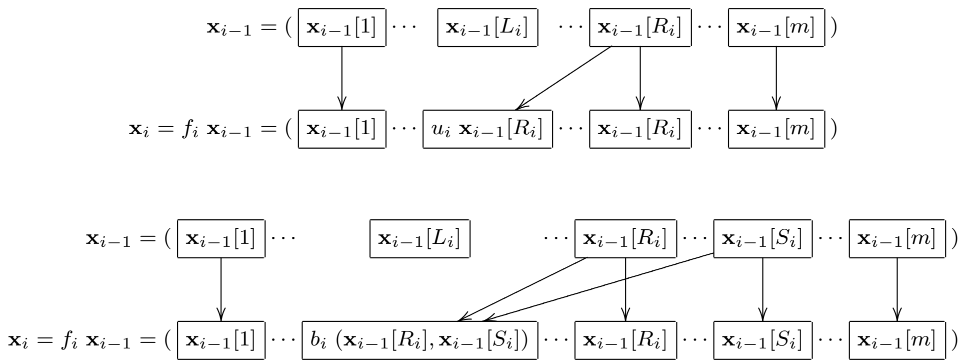


Fig. 1. Graphical depiction of unary and binary machine-state transition functions.

be resolved, in the first case, by computing $\overrightarrow{\mathbf{x}_n} = (\mathcal{J}\ f\ \mathbf{x}_0) \times \overrightarrow{\mathbf{x}_0}$:

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

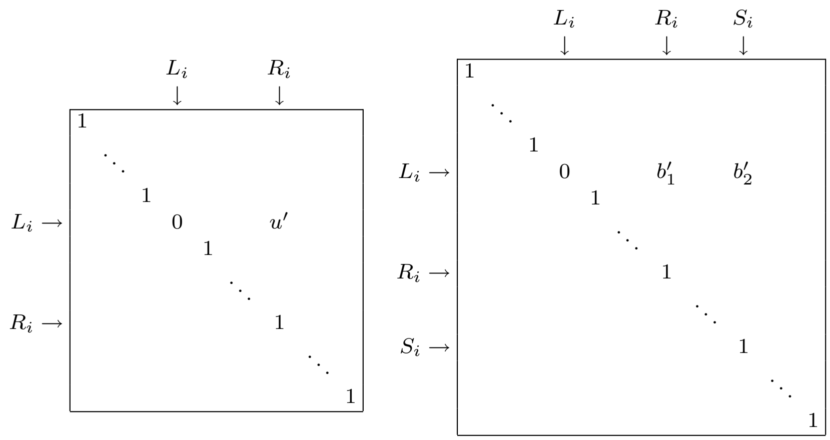
and, in the second case, by computing $\overleftarrow{\mathbf{x}_0} = (\mathcal{J}\ f\ \mathbf{x}_0)^\top \times \overleftarrow{\mathbf{x}_n}$:

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

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 $\overrightarrow{\mathbf{x}_i}$ as the *perturbation* variables, and the $\overleftarrow{\mathbf{x}_i}$ as the *sensitivity* variables. The k-th column of the Jacobian $\overrightarrow{\mathbf{X}_n}$ can be recovered by taking $\overrightarrow{\mathbf{x}_0}$ to be a basis vector with a one at index k and zeros elsewhere and computing $\overrightarrow{\mathbf{x}_n}$. Correspondingly, the j-th row can be recovered by taking $\overleftarrow{\mathbf{x}_n}$ to be a basis vector with a one at index j and zeros elsewhere and computing $\overleftarrow{\mathbf{x}_0}$.

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

The transition functions f_i typically compute a single element of $\mathbf{x}_i$ at index L_i, either as a unary scalar function u_i of a single element of $\mathbf{x}_{i-1}$ at index R_i or as a binary scalar function b_i of two elements of $\mathbf{x}_{i-1}$ at indices R_i and S_i passing the remaining elements of $\mathbf{x}_{i-1}$ unchanged through to $\mathbf{x}_i$. We refer to such functions as unary and binary machine-state transition functions (Figure 1). (Another way



$u' = \mathcal{D}\ u_i\ \mathbf{x}_{i-1}[R_i]$

$b'_1 = \mathcal{D}_1\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i])$
$b'_2 = \mathcal{D}_2\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i])$

Fig. 2. The Jacobian $\mathcal{J}\ f_i\ \mathbf{x}_{i-1}$ of unary and binary machine-state transition functions.

of viewing this is that we have n assignments, each of the form $\mathbf{x}[L_i] := u_i\ \mathbf{x}[R_i]$ or $\mathbf{x}[L_i] := b\ (\mathbf{x}[R_i], \mathbf{x}[S_i])$, where we use $i = 1, \ldots, n$ to index assignments.) In this case, the Jacobian matrices $\mathcal{J}\ f_i\ \mathbf{x}_{i-1}$ are sparse, differing from the identity matrix by only a few elements (Figure 2). Because of this, $\overrightarrow{\mathbf{x}_i}$ differs from $\overrightarrow{\mathbf{x}_{i-1}}$ at only a few elements (Figure 3) and $\overleftarrow{\mathbf{x}_{i-1}}$ differs from $\overleftarrow{\mathbf{x}_i}$ only at a few elements (Figure 4).

When computing the sensitivities (Figure 4), note that the sensitivities for the input indices R_i and S_i are accumulated and the sensitivity for the output index L_i is zeroed. This explains why fanout (more than one use of a computed quantity) in the original computation necessitates addition during the computation of sensitivities, and destructive assignment (overwriting a computed quantity) necessitates zeroing of a sensitivity. Figures 1–4 assume that all indices are distinct. When the indices are not distinct, the structure of the Jacobian (Figure 2) simplifies. This introduces some additional cases in the following analysis.

If we let $u' = \mathcal{D}\ u_i\ \mathbf{x}_{i-1}[R_i]$, in the unary case:

$$\begin{aligned}
\mathbf{x}_i[j] &= (f_i\ \mathbf{x}_{i-1})[j] \\
&= \begin{cases} u_i\ \mathbf{x}_{i-1}[R_i] & \text{when } j = L_i \\ \mathbf{x}_{i-1}[j] & \text{otherwise} \end{cases} \\
(\mathcal{J}\ f_i\ \mathbf{x}_{i-1})[j,k] &= \begin{cases} u' & \text{when } j = L_i \wedge k = R_i \\ 1 & \text{when } j \neq L_i \wedge j = k \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$
\begin{pmatrix}
\overrightarrow{\mathbf{x}_{i-1}}[1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i-1] \\
u' \times \overrightarrow{\mathbf{x}_{i-1}}[R_i] \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i+1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[R_i] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[m]
\end{pmatrix}
=
\begin{pmatrix}
1 & & & & & & & \\
 & \ddots & & & & & & \\
 & & 1 & & & & & \\
 & & & 0 & & & u' & \\
 & & & & 1 & & & \\
 & & & & & \ddots & & \\
 & & & & & & 1 & \\
 & & & & & & & \ddots & \\
 & & & & & & & & 1
\end{pmatrix}
\begin{pmatrix}
\overrightarrow{\mathbf{x}_{i-1}}[1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i-1] \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i] \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i+1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[R_i] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[m]
\end{pmatrix}
$$

$$u' = \mathcal{D}\ u_i\ \mathbf{x}_{i-1}[R_i]$$

$$
\begin{pmatrix}
\overrightarrow{\mathbf{x}_{i-1}}[1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i-1] \\
b'_1 \times \overrightarrow{\mathbf{x}_{i-1}}[R_i] + b'_2 \times \overrightarrow{\mathbf{x}_{i-1}}[S_i] \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i+1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[R_i] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[S_i] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[m]
\end{pmatrix}
=
\begin{pmatrix}
1 & & & & & & & & & & \\
 & \ddots & & & & & & & & & \\
 & & 1 & & & & & & & & \\
 & & & 0 & & & b'_1 & & b'_2 & & \\
 & & & & 1 & & & & & & \\
 & & & & & \ddots & & & & & \\
 & & & & & & 1 & & & & \\
 & & & & & & & \ddots & & & \\
 & & & & & & & & 1 & & \\
 & & & & & & & & & \ddots & \\
 & & & & & & & & & & 1
\end{pmatrix}
\begin{pmatrix}
\overrightarrow{\mathbf{x}_{i-1}}[1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i-1] \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i] \\
\overrightarrow{\mathbf{x}_{i-1}}[L_i+1] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[R_i] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[S_i] \\
\vdots \\
\overrightarrow{\mathbf{x}_{i-1}}[m]
\end{pmatrix}
$$

$$
\begin{aligned}
b'_1 &= \mathcal{D}_1\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i]) \\
b'_2 &= \mathcal{D}_2\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i])
\end{aligned}
$$

Fig. 3. Forward-mode AD: computing $\overrightarrow{\mathbf{x}_i} = (\mathcal{J}\ f_i\ \mathbf{x}_{i-1}) \times \overrightarrow{\mathbf{x}_{i-1}}$ for unary and binary machine-state transition functions.

$$
\begin{aligned}
\overrightarrow{\mathbf{x}_i}[j] &= ((\mathcal{J}\ f_i\ \mathbf{x}_{i-1}) \times \overrightarrow{\mathbf{x}_{i-1}})[j] \\
&= \begin{cases} u' \times \overrightarrow{\mathbf{x}_{i-1}}[R_i] & \text{when } j = L_i \\ \overrightarrow{\mathbf{x}_{i-1}}[j] & \text{otherwise} \end{cases} \\
\overleftarrow{\mathbf{x}_{i-1}}[k] &= ((\mathcal{J}\ f_i\ \mathbf{x}_{i-1})^\top \times \overleftarrow{\mathbf{x}_i})[k] \\
&= \begin{cases} \overleftarrow{\mathbf{x}_i}[k] + u' \times \overleftarrow{\mathbf{x}_i}[L_i] & \text{when } k = R_i \wedge R_i \neq L_i \\ u' \times \overleftarrow{\mathbf{x}_i}[k] & \text{when } k = R_i = L_i \\ 0 & \text{when } k \neq R_i \wedge k = L_i \\ \overleftarrow{\mathbf{x}_i}[k] & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{pmatrix}
\overleftarrow{\mathbf{x}}_i[1] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[L_i-1] \\
0 \\
\overleftarrow{\mathbf{x}}_i[L_i+1] \\
\vdots \\
u' \times \overleftarrow{\mathbf{x}}_i[L_i] + \overleftarrow{\mathbf{x}}_i[R_i] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[m]
\end{pmatrix}
=
\begin{pmatrix}
1 &&&&&&& \\
& \ddots &&&&&& \\
&& 1 &&&&& \\
&&& 0 &&&& \\
&&&& 1 &&& \\
&&&&& \ddots && \\
&&& u' &&& 1 & \\
&&&&&&& \ddots \\
&&&&&&&& 1
\end{pmatrix}
\begin{pmatrix}
\overleftarrow{\mathbf{x}}_i[1] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[L_i-1] \\
\overleftarrow{\mathbf{x}}_i[L_i] \\
\overleftarrow{\mathbf{x}}_i[L_i+1] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[R_i] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[m]
\end{pmatrix}
$$

$$u' = \mathcal{D}\ u_i\ \mathbf{x}_{i-1}[R_i]$$

$$
\begin{pmatrix}
\overleftarrow{\mathbf{x}}_i[1] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[L_i-1] \\
0 \\
\overleftarrow{\mathbf{x}}_i[L_i+1] \\
\vdots \\
b'_1 \times \overleftarrow{\mathbf{x}}_i[L_i] + \overleftarrow{\mathbf{x}}_i[R_i] \\
\vdots \\
b'_2 \times \overleftarrow{\mathbf{x}}_i[L_i] + \overleftarrow{\mathbf{x}}_i[S_i] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[m]
\end{pmatrix}
=
\begin{pmatrix}
1 &&&&&&&&& \\
& \ddots &&&&&&&& \\
&& 1 &&&&&&& \\
&&& 0 &&&&&& \\
&&&& 1 &&&&& \\
&&&&& \ddots &&&& \\
&&& b'_1 &&& 1 &&& \\
&&&&&&& \ddots && \\
&&& b'_2 &&&&& 1 & \\
&&&&&&&&& \ddots \\
&&&&&&&&&& 1
\end{pmatrix}
\begin{pmatrix}
\overleftarrow{\mathbf{x}}_i[1] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[L_i-1] \\
\overleftarrow{\mathbf{x}}_i[L_i] \\
\overleftarrow{\mathbf{x}}_i[L_i+1] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[R_i] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[S_i] \\
\vdots \\
\overleftarrow{\mathbf{x}}_i[m]
\end{pmatrix}
$$

$$
\begin{aligned}
b'_1 &= \mathcal{D}_1\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i]) \\
b'_2 &= \mathcal{D}_2\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i])
\end{aligned}
$$

Fig. 4. Reverse-mode AD: computing $\overleftarrow{\mathbf{x}_{i-1}} = (\mathcal{J}\ f_i\ \mathbf{x}_{i-1})^\top \times \overleftarrow{\mathbf{x}_i}$ for unary and binary machine-state transition functions.

If we let $b'_1 = \mathcal{D}_1\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i])$ and $b'_2 = \mathcal{D}_2\ b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i])$, in the binary case:

$$
\begin{aligned}
\mathbf{x}_i[j] &= (f_i\ \mathbf{x}_{i-1})[j] \\
&= \begin{cases} b_i\ (\mathbf{x}_{i-1}[R_i], \mathbf{x}_{i-1}[S_i]) & \text{when } j = L_i \\ \mathbf{x}_{i-1}[j] & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
(\mathcal{J}\ f_i\ \mathbf{x}_{i-1})[j,k] = \begin{cases}
b'_1 & \text{when } j = L_i \wedge k = R_i \wedge R_i \neq S_i \\
b'_2 & \text{when } j = L_i \wedge k = S_i \wedge R_i \neq S_i \\
b'_1 + b'_2 & \text{when } j = L_i \wedge k = R_i = S_i \\
1 & \text{when } j \neq L_i \wedge j = k \\
0 & \text{otherwise}
\end{cases}
$$

$$
\begin{aligned}
\acute{\mathbf{x}}_i[j] &= ((\mathcal{J}\ f_i\ \mathbf{x}_{i-1}) \times \acute{\mathbf{x}}_{i-1})[j] \\
&= \begin{cases} b'_1 \times \acute{\mathbf{x}}_{i-1}[R_i] + b'_2 \times \acute{\mathbf{x}}_{i-1}[S_i] & \text{when } j = L_i \\ \acute{\mathbf{x}}_{i-1}[j] & \text{otherwise} \end{cases} \\
\grave{\mathbf{x}}_{i-1}[k] &= ((\mathcal{J}\ f_i\ \mathbf{x}_{i-1})^\top \times \grave{\mathbf{x}}_i)[k] \\
&= \begin{cases}
\grave{\mathbf{x}}_i[k] + b'_1 \times \grave{\mathbf{x}}_i[L_i] & \text{when } k = R_i \wedge R_i \neq S_i \wedge R_i \neq L_i \\
\grave{\mathbf{x}}_i[k] + b'_2 \times \grave{\mathbf{x}}_i[L_i] & \text{when } k = S_i \wedge R_i \neq S_i \wedge S_i \neq L_i \\
b'_1 \times \grave{\mathbf{x}}_i[k] & \text{when } k = R_i = L_i \wedge R_i \neq S_i \\
b'_2 \times \grave{\mathbf{x}}_i[k] & \text{when } k = S_i = L_i \wedge R_i \neq S_i \\
\grave{\mathbf{x}}_i[k] + (b'_1 + b'_2) \times \grave{\mathbf{x}}_i[L_i] & \text{when } k = R_i = S_i \wedge k \neq L_i \\
(b'_1 + b'_2) \times \grave{\mathbf{x}}_i[k] & \text{when } k = R_i = S_i = L_i \\
0 & \text{when } k \neq R_i \wedge k \neq S_i \wedge k = L_i \\
\grave{\mathbf{x}}_i[k] & \text{otherwise}
\end{cases}
\end{aligned}
$$

With forward-mode AD, computation of the perturbation variables $\acute{\mathbf{x}}_i$ can be interleaved with the original primal computation:

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

This leads to a simple transformation:

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

with an appropriate definition of $\overrightarrow{\cdot}$ on vectors and functions thereof:

$$
\begin{aligned}
\overrightarrow{\mathbf{x}} &\equiv (\mathbf{x}, \acute{\mathbf{x}}) \\
\overrightarrow{f}\ \overrightarrow{\mathbf{x}} &\equiv ((f\ \mathbf{x}), ((\mathcal{J}\ f\ \mathbf{x}) \times \acute{\mathbf{x}}))
\end{aligned}
$$

The fact that $\mathbf{x}_{i-1}$ and $\acute{\mathbf{x}}_{i-1}$ are no longer referenced once $\mathbf{x}_i$ and $\acute{\mathbf{x}}_i$ 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 $\acute{\mathbf{x}}$, means that when the f_i are machine-state transition functions, the original program can be written as a sequence of assignments of the form $x_{L_i} := u_i\ x_{R_i}$ and $x_{L_i} := b_i\ (x_{R_i}, x_{S_i})$, referencing variables x that contain scalars, and the transformed program can be written as a sequence of assignments of the form $\overrightarrow{x_{L_i}} := \overrightarrow{u_i}\ \overrightarrow{x_{R_i}}$ and $\overrightarrow{x_{L_i}} := \overrightarrow{b_i}\ (\overrightarrow{x_{R_i}}, \overrightarrow{x_{S_i}})$, referencing variables $\overrightarrow{x} \equiv (x, \acute{x})$ where:

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

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 $\acute{x}$ and by overloading the primitives u and b with $\overrightarrow{u}$ and $\overrightarrow{b}$ respectively. This implementation technique obviously preserves both the temporal and spatial complexity of the program. In the functional realm, forward-mode AD has been implemented in HASKELL [Karczmarczuk, 1998a,b, 1999, 2001b] and SCHEME [Sussman et al., 2001; Siskind and Pearlmutter, 2007; Pearlmutter and Siskind, 2007], although not all of these systems support nesting [Siskind and Pearlmutter, 2005].

In contrast, with reverse-mode AD, computation of the sensitivity variables $\grave{\mathbf{x}}_i$ *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:

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

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 $\grave{\mathbf{x}}_i$ 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 *non-local* program transformation.

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

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

If we take $\bar{\mathbf{x}}_0$ to be the identity function, the reverse phase can be performed by evaluating $\bar{\mathbf{x}}_n\ \grave{\mathbf{x}}_n$. We refer to $\bar{\mathbf{x}}$ 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:

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

with an appropriate definition of $\overleftarrow{\cdot}$ on vectors and functions thereof:

$$\begin{aligned}\overleftarrow{\mathbf{x}} &\equiv (\mathbf{x}, \overline{\mathbf{x}})\\ \overleftarrow{f}\ \overleftarrow{\mathbf{x}} &\equiv ((f\ \mathbf{x}), (\lambda\,\grave{\mathbf{x}}\ \ \overline{\mathbf{x}}\ ((\mathcal{J}\ f\ \mathbf{x})^\top \times \grave{\mathbf{x}})))\end{aligned}$$

However unlike in forward mode, where $\overrightarrow{\mathbf{x}} \equiv (\mathbf{x}, \acute{\mathbf{x}})$ can be interleaved as a vector of pairs, it is not possible to interleave $\overleftarrow{\mathbf{x}} \equiv (\mathbf{x}, \overline{\mathbf{x}})$ because $\overline{\mathbf{x}}$ is a function rather than a vector. Thus, one must use different techniques to implement reverse-mode AD with a local program transformation that takes advantage of the locality of machine-state transition functions.

The traditional way this is done is to maintain a single global backpropagator variable $\overline{x}$ that is updated via side effect and by taking:

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

This eliminates the need to pair backpropagators with primal values and allows taking $\overleftarrow{\mathbf{x}} \equiv \mathbf{x}$. When the f_i are machine-state transition functions, and the original program is written as a sequence of assignments of the form $x_{L_i} := u_i\ x_{R_i}$ and $x_{L_i} := b_i\ (x_{R_i}, x_{S_i})$, referencing variables x that contain scalars, the transformed program can be written as a sequence of assignments[4] of the form:

$$\begin{aligned}\overline{x} := \lambda[\,]\ \mathbf{begin}\ &\grave{x}_{R_i} \mathrel{+}:= (\mathcal{D}\ u_i\ \overleftarrow{x_{R_i}}) \times \grave{x}_{L_i};\\ &\grave{x}_{L_i} := 0;\\ &\overline{x}\ [\,]\ \mathbf{end}\\ \overleftarrow{x_{L_i}} := u_i\ \overleftarrow{x_{R_i}}\end{aligned}$$

[4]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 $\grave{\mathbf{x}}_n$, initializing all other sensitivity variables to zero, calling the backpropagator $\overline{x}$, and examining the values of $\grave{\mathbf{x}}_0$ that remain in select sensitivity variables after the backpropagator returns.

and:

$$
\begin{aligned}
\overline{x} := \lambda[\,]\ \textbf{begin}\ & \overleftharpoon{x_{R_i}} \mathrel{+}:= (\mathcal{D}_1\ b_i\ (\overleftarrow{x_{R_i}}, \overleftarrow{x_{S_i}})) \times \overleftharpoon{x_{L_i}};\\
& \overleftharpoon{x_{S_i}} \mathrel{+}:= (\mathcal{D}_2\ b_i\ (\overleftarrow{x_{R_i}}, \overleftarrow{x_{S_i}})) \times \overleftharpoon{x_{L_i}};\\
& \overleftharpoon{x_{L_i}} := 0;\\
& \overline{x}\ [\,]\ \textbf{end}\\
\overleftarrow{x_{L_i}} := b_i\ (\overleftarrow{x_{R_i}}, \overleftarrow{x_{S_i}})
\end{aligned}
$$

taking $\overleftarrow{x} \equiv x$. The above transformation assumes that the indices are distinct, i.e., $L_i \neq R_i \neq S_i$, but it is straightforward to relax this assumption.

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 non-local program transformation. One simply postpends the program with assignments to initialize the sensitivity variables and then postpends assignments of the form:

$$
\begin{aligned}
&\overleftharpoon{x_{R_i}} \mathrel{+}:= (\mathcal{D}\ u_i\ \overleftarrow{x_{R_i}}) \times \overleftharpoon{x_{L_i}};\\
&\overleftharpoon{x_{L_i}} := 0
\end{aligned}
$$

for each primal assignment $x_{L_i} := u_i\ x_{R_i}$, and of the form:

$$
\begin{aligned}
&\overleftharpoon{x_{R_i}} \mathrel{+}:= (\mathcal{D}_1\ b_i\ (\overleftarrow{x_{R_i}}, \overleftarrow{x_{S_i}})) \times \overleftharpoon{x_{L_i}};\\
&\overleftharpoon{x_{S_i}} \mathrel{+}:= (\mathcal{D}_2\ b_i\ (\overleftarrow{x_{R_i}}, \overleftarrow{x_{S_i}})) \times \overleftharpoon{x_{L_i}};\\
&\overleftharpoon{x_{L_i}} := 0
\end{aligned}
$$

for each primal assignment $x_{L_i} := b_i\ (x_{R_i}, x_{S_i})$, in reverse order to their occurrence in the primal. This again assumes that $L_i \neq R_i \neq S_i$.

Note that reverse-mode AD preserves the temporal complexity of the program. However due to the need to save primal values for use during the reverse phase, spatial complexity is not preserved. 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:

$$
\begin{aligned}
x_{L_1} &:= u_1\ x_{S_1}\\
&\ \vdots\\
x_{L_n} &:= u_n\ x_{S_n}
\end{aligned}
$$

From Section 2, the tapeless non-local reverse-mode AD transformation of this program is:

$$\left.\begin{array}{c} x_{L_1} := u_1\ x_{S_1} \\ \vdots \\ x_{L_n} := u_n\ x_{S_n} \end{array}\right\} \leadsto \left\{\begin{array}{l} x_{L_1} := u_1\ x_{S_1} \\ \qquad\vdots \\ x_{L_n} := u_n\ x_{S_n} \\ \overleftarrow{x_1} := 0 \\ \qquad\vdots \\ \overleftarrow{x_m} := 0 \\ \overleftarrow{x_{S_n}} \mathrel{+}:= (\mathcal{D}\ u_n\ x_{S_n}) \times \overleftarrow{x_{L_n}} \\ \overleftarrow{x_{L_n}} := 0 \\ \qquad\vdots \\ \overleftarrow{x_{S_1}} \mathrel{+}:= (\mathcal{D}\ u_1\ x_{S_1}) \times \overleftarrow{x_{L_1}} \\ \overleftarrow{x_{L_1}} := 0 \end{array}\right.$$

Care must be taken in the above to omit the initialization $\overleftarrow{x_j} := 0$ for $j = L_n$.

If we restrict our consideration to single-assignment code, the left-hand sides x_{L_i} of the assignments can be replaced (by alpha renaming) with x_i. In such single-assignment code, the special cases considered in the previous section to deal with the possibility that the same variable appears on both the left- and right-hand sides of an assignment are not needed. Furthermore, the zeroing assignments $\overleftarrow{x_{L_i}} := 0$ during the reverse phase that would result from destructive assignments in the forward phase can be eliminated. We typographically distinguish single-assignment code fragments from destructive-assignment code fragments by denoting the former with $=$ and the latter with $:=$. Note that even though the forward phase is now single assignment the reverse phase is not, due to fanout in the forward phase.

$$\left.\begin{array}{c} x_1 = u_1\ x_{S_1} \\ \vdots \\ x_n = u_n\ x_{S_n} \end{array}\right\} \leadsto \left\{\begin{array}{l} x_1 = u_1\ x_{S_1} \\ \qquad\vdots \\ x_n = u_n\ x_{S_n} \\ \overleftarrow{x_0} := 0 \\ \qquad\vdots \\ \overleftarrow{x_{n-1}} := 0 \\ \overleftarrow{x_{S_n}} \mathrel{+}:= (\mathcal{D}\ u_n\ x_{S_n}) \times \overleftarrow{x_n} \\ \qquad\vdots \\ \overleftarrow{x_{S_1}} \mathrel{+}:= (\mathcal{D}\ u_1\ x_{S_1}) \times \overleftarrow{x_1} \end{array}\right.$$

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

$$\left.\begin{array}{c} x_1 = u_1\ x_{S_1} \\ \vdots \\ x_n = u_n\ x_{S_n} \end{array}\right\} \rightsquigarrow \left\{\begin{array}{c} x_1 = u_1\ x_{S_1} \\ \vdots \\ x_n = u_n\ x_{S_n} \\ \grave{x_0} := 0 \\ \vdots \\ \grave{x_{n-1}} := 0 \\ \grave{x_{S_n}} \mathrel{+}:= \overline{u_n}\ \grave{x_n} \\ \vdots \\ \grave{x_{S_1}} \mathrel{+}:= \overline{u_1}\ \grave{x_1} \end{array}\right.$$

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

$$\left.\begin{array}{c} x_1 = u_1\ x_{S_1} \\ \vdots \\ x_n = u_n\ x_{S_n} \end{array}\right\} \rightsquigarrow \left\{\begin{array}{c} (x_1, \overline{x_1}) = \overleftarrow{u_1}\ x_{S_1} \\ \vdots \\ (x_n, \overline{x_n}) = \overleftarrow{u_n}\ x_{S_n} \\ \grave{x_0} := 0 \\ \vdots \\ \grave{x_{n-1}} := 0 \\ \grave{x_{S_n}} \mathrel{+}:= \overline{x_n}\ \grave{x_n} \\ \vdots \\ \grave{x_{S_1}} \mathrel{+}:= \overline{x_1}\ \grave{x_1} \end{array}\right.$$

Note that this transformation is valid only for single-assignment code. The backpropagator variables $\overline{x_i}$ 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.

Here, each $\overline{x_i}$ is simply a function from $\grave{x_i}$ to $\grave{x_{S_i}}$. Evaluating $\overline{x_i}\ \grave{x_i}$ has the same temporal complexity as evaluating $u_i\ x_{S_i}$. 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:

$$\left.\begin{array}{c} x_1 = x_{R_1}\ x_{S_1} \\ \vdots \\ x_n = x_{R_n}\ x_{S_n} \end{array}\right\} \rightsquigarrow \left\{\begin{array}{r@{}l} (\overleftarrow{x_1}, \overline{x_1}) &= \overleftarrow{x_{R_1}}\ \overleftarrow{x_{S_1}} \\ &\vdots \\ (\overleftarrow{x_n}, \overline{x_n}) &= \overleftarrow{x_{R_n}}\ \overleftarrow{x_{S_n}} \\ \overleftharpoon{x_0} &:= 0 \\ &\vdots \\ \overleftharpoon{x_{n-1}} &:= 0 \\ \overleftharpoon{x_{S_n}} &\mathrel{+}:= \overline{x_n}\ \overleftharpoon{x_n} \\ &\vdots \\ \overleftharpoon{x_{S_1}} &\mathrel{+}:= \overline{x_1}\ \overleftharpoon{x_1} \end{array}\right.$$

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 $\overleftharpoon{x}$ to contain arbitrary transformed and sensitivity values that correspond to the arbitrary untransformed values stored in the corresponding variables x. This requires us to add arbitrary sensitivity values.

We define some new machinery to facilitate manipulation of sensitivities of potentially aggregate data. Loosely speaking, the first is a unary function $\mathbf{0}$ that maps any (potentially aggregate) value to an otherwise identical value except that all reals have been replaced with zeros. The second is a binary function $\oplus$ which can only be invoked on (potentially aggregate) conformant values x_1 and x_2, where by conformant we mean that $(\mathbf{0}\ x_1) = (\mathbf{0}\ x_2)$. The value $x_3 = x_1 \oplus x_2$ conforms to x_1 and x_2, differing only in that the real values in x_3 are the sum of the real values at corresponding positions in x_1 and x_2. If we regard (potentially nested) aggregate values as scaffolding for the vector of reals at their fringe, $\oplus$ is simply vector addition and $\mathbf{0}$ can be viewed either as scalar multiplication by zero or as a constructor for the zero vector of this vector space. Given this view, one can generalize the notions of vectors, matrices, Jacobians, matrix transposition, and matrix-vector multiplication, and thus forward- and reverse-mode AD, to arbitrary aggregate data. Finally, we define a unary function $\overleftarrow{\mathcal{J}}$ that maps values to corresponding transformed values and its inverse function $\overleftarrow{\mathcal{J}}^{-1}$ that maps transformed values to corresponding untransformed values. This machinery allows $\overleftharpoon{x}$ to be properly initialized with $\mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftharpoon{x})$:

$$
\left.\begin{array}{c}
x_1 = x_{R_1}\ x_{S_1}\\
\vdots\\
x_n = x_{R_n}\ x_{S_n}
\end{array}\right\}
\rightsquigarrow
\left\{\begin{array}{r@{\ }l}
(\overleftharpoon{x_1}, \overline{x_1}) &= \overleftharpoon{x_{R_1}}\ \overleftharpoon{x_{S_1}}\\
&\vdots\\
(\overleftharpoon{x_n}, \overline{x_n}) &= \overleftharpoon{x_{R_n}}\ \overleftharpoon{x_{S_n}}\\
\grave{x_0} &:= \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftharpoon{x_0})\\
&\vdots\\
\grave{x_{n-1}} &:= \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftharpoon{x_{n-1}})\\
\grave{x_{S_n}} &\oplus\!:= \overline{x_n}\ \grave{x_n}\\
&\vdots\\
\grave{x_{S_1}} &\oplus\!:= \overline{x_1}\ \grave{x_1}
\end{array}\right.
$$

The above transformation of single-assignment straight-line code can be applied to transform any α-converted lambda expression in A-normal form [Sabry and Felleisen, 1993]. (A lambda expression is in A-normal form if its body is a `let*` whose body is a variable reference and whose binding expressions are all variable references, applications of variable references to variable references, or lambda expressions in A-normal form. Lambda expressions in A-normal form are thus analogous to straight-line code. The requirement for α-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 $\grave{x_n}$ to $\grave{x_0}$, by multiplying the transpose of the Jacobian of the function that maps x_0 to x_n, at x_0, by $\grave{x_n}$, under appropriate generalizations of the notions of vectors, matrices, Jacobians, matrix transposition, and matrix-vector multiplication. 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:

$$
\left.\begin{array}{l}
\lambda x_0\ \textbf{let}\ x_1 \triangleq x_{R_1}\ x_{S_1};\\
\qquad\vdots\\
\qquad x_n \triangleq x_{R_n}\ x_{S_n}\\
\quad\textbf{in}\ x_n\ \textbf{end}
\end{array}\right\}
\leadsto
\left\{\begin{array}{l}
\lambda\overleftarrow{x_0}\ \textbf{let}\ (\overleftarrow{x_1},\overline{x_1}) \triangleq \overleftarrow{x_{R_1}}\ \overleftarrow{x_{S_1}};\\
\qquad\qquad\vdots\\
\qquad\qquad(\overleftarrow{x_n},\overline{x_n}) \triangleq \overleftarrow{x_{R_n}}\ \overleftarrow{x_{S_n}}\\
\qquad\textbf{in}\ (\overleftarrow{x_n},(\lambda\overleftharpoon{x_n}\ \textbf{let}\ \overleftharpoon{x_0} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_0});\\
\qquad\qquad\qquad\qquad\vdots\\
\qquad\qquad\qquad\qquad\overleftharpoon{x_{n-1}} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_{n-1}});\\
\qquad\qquad\qquad\qquad\overleftharpoon{x_{S_n}} \oplus\!\!\triangleq \overline{x_n}\ \overleftharpoon{x_n};\\
\qquad\qquad\qquad\qquad\vdots\\
\qquad\qquad\qquad\qquad\overleftharpoon{x_{S_1}} \oplus\!\!\triangleq \overline{x_1}\ \overleftharpoon{x_1}\\
\qquad\qquad\qquad\textbf{in}\ \overleftharpoon{x_0}\ \textbf{end}))\ \textbf{end}
\end{array}\right.
$$

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 the 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:

$$
\begin{array}{l}
f \triangleq \lambda a\ \textbf{let}\ x_1 \triangleq \lambda b\ \textbf{let}\ x_4 \triangleq \lambda c\ b\\
\qquad\qquad\qquad\qquad\qquad\textbf{in}\ x_4\ \textbf{end};\\
\qquad\qquad x_2 \triangleq x_1\ a;\\
\qquad\qquad x_3 \triangleq x_2\ 1 \qquad\qquad \texttt{/*}\mathrm{I}\texttt{*/}\\
\qquad\quad\textbf{in}\ x_3\ \textbf{end}
\end{array}
$$

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

$$
\begin{array}{l}
\overleftarrow{f} \triangleq \lambda\overleftarrow{a}\ \textbf{let}\ \overleftarrow{x_1} \triangleq \lambda\overleftarrow{b}\ \textbf{let}\ \overleftarrow{x_4} \triangleq \lambda\overleftarrow{c}\ (\overleftarrow{b},(\lambda\overleftharpoon{b}\ \overleftharpoon{c}))\ \ \texttt{/*}\mathrm{II}\texttt{*/}\\
\qquad\qquad\qquad\qquad\qquad\textbf{in}\ (\overleftarrow{x_4},(\lambda\overleftharpoon{x_4}\ \textbf{let}\ \overleftharpoon{b} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{b})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textbf{in}\ \overleftharpoon{b}\ \textbf{end}))\ \textbf{end};\\
\qquad\qquad(\overleftarrow{x_2},\overline{x_2}) \triangleq \overleftarrow{x_1}\ \overleftarrow{a};\\
\qquad\qquad(\overleftarrow{x_3},\overline{x_3}) \triangleq \overleftarrow{x_2}\ \overleftarrow{1} \qquad\qquad\qquad\qquad \texttt{/*}\mathrm{III}\texttt{*/}\\
\qquad\textbf{in}\ (\overleftarrow{x_3},(\lambda\overleftharpoon{x_3}\ \textbf{let}\ \overleftharpoon{a} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{a});\\
\qquad\qquad\qquad\qquad\overleftharpoon{x_1} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_1});\\
\qquad\qquad\qquad\qquad\overleftharpoon{x_2} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_2});\\
\qquad\qquad\qquad\qquad\overleftharpoon{1} \oplus\!\!\triangleq \overline{x_3}\ \overleftharpoon{x_3}; \qquad\qquad \texttt{/*}\mathrm{IV}\texttt{*/}\\
\qquad\qquad\qquad\qquad\overleftharpoon{a} \oplus\!\!\triangleq \overline{x_2}\ \overleftharpoon{x_2}\\
\qquad\qquad\qquad\textbf{in}\ \overleftharpoon{a}\ \textbf{end}))\ \textbf{end}
\end{array}
$$

The above code is trivially incorrect, because there are references to unbound variables $\overleftharpoon{c}$, $\overleftarrow{1}$, and $\overleftharpoon{1}$ in lines II, III, and IV. The free reference to $\overleftarrow{1}$ in line III results from transforming the constant 1 in line I 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 III but does not address the free references to $\grave{c}$ and $\grave{1}$ in lines II and IV. 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 $\grave{c}$ and $\grave{1}$ in lines V and VI below:

$$
\begin{array}{l}
\overleftarrow{f} \triangleq \lambda \overleftarrow{a}\ \textbf{let}\ \overleftarrow{x_1} \triangleq \lambda \overleftarrow{b}\ \textbf{let}\ \overleftarrow{x_4} \triangleq \lambda \overleftarrow{c}\ (\overleftarrow{b}, (\lambda \grave{b}\ \textbf{let}\ \grave{c} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{c}) \quad \texttt{/*}\mathrm{V}\texttt{*/} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \grave{c}\ \textbf{end})) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\overleftarrow{x_4}, (\lambda \grave{x_4}\ \textbf{let}\ \grave{b} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{b}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \grave{b}\ \textbf{end}))\ \textbf{end}; \\
\qquad\qquad (\overleftarrow{x_2}, \overline{x_2}) \triangleq \overleftarrow{x_1}\ \overleftarrow{a}; \\
\qquad\qquad (\overleftarrow{x_3}, \overline{x_3}) \triangleq \overleftarrow{x_2}\ \overleftarrow{1} \\
\qquad \textbf{in}\ (\overleftarrow{x_3}, (\lambda \grave{x_3}\ \textbf{let}\ \grave{a} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{a}); \\
\qquad\qquad\qquad\qquad\qquad \grave{x_1} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_1}); \\
\qquad\qquad\qquad\qquad\qquad \grave{x_2} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_2}); \\
\qquad\qquad\qquad\qquad\qquad \grave{1} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{1}); \quad \texttt{/*}\mathrm{VI}\texttt{*/} \\
\qquad\qquad\qquad\qquad\qquad \grave{1} \oplus\!\!\triangleq \overline{x_3}\ \grave{x_3}; \\
\qquad\qquad\qquad\qquad\qquad \grave{a} \oplus\!\!\triangleq \overline{x_2}\ \grave{x_2} \\
\qquad\qquad\qquad\quad \textbf{in}\ \grave{a}\ \textbf{end}))\ \textbf{end}
\end{array}
$$

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, $\grave{x_3}$ is bound to 1. Then, $\grave{a}$, $\grave{x_1}$, $\grave{x_2}$, and $\grave{1}$ are bound to zeros. Then, we apply $\overline{x_3}$ to 1. Since $\overline{x_3}$ is bound to $\lambda \grave{b} \ldots$, $\grave{b}$ is bound to 1, $\grave{c}$ is bound to a zero, $\lambda \grave{b} \ldots$ returns a zero, and $\grave{1}$ is incremented by a zero and remains a zero. Then, we apply $\overline{x_2}$ to a zero. Since $\overline{x_2}$ is bound to $\lambda \grave{x_4} \ldots$, $\grave{x_4}$ is bound to a zero, $\grave{b}$ is bound to a zero, $\lambda \grave{x_4} \ldots$ returns a zero, and $\grave{a}$ 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 VII, IX, and XIII 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 X and XI below. Finally, lines are generated in backpropagators that correspond to nested lambda expressions in the untransformed function, as shown in lines VIII and XII below:

$$
\begin{array}{l}
\overleftarrow{f} \stackrel{\triangle}{=} \lambda \overleftarrow{a}\ \textbf{let}\ \overleftarrow{x_1} \stackrel{\triangle}{=} \lambda \overleftarrow{b}\ \textbf{let}\ \overleftarrow{x_4} \stackrel{\triangle}{=} \lambda \overleftarrow{c}\ (\overleftarrow{b}, (\lambda \overset{\leftharpoondown}{b}\ \textbf{let}\ \overset{\leftharpoondown}{c} \stackrel{\triangle}{=} \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{c}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ ([\overset{\leftharpoondown}{b}], \overset{\leftharpoondown}{c})\ \textbf{end})) \quad \texttt{/*VII*/} \\
\qquad\qquad\qquad\qquad \textbf{in}\ (\overleftarrow{x_4}, (\lambda \overset{\leftharpoondown}{x_4}\ \textbf{let}\ \overset{\leftharpoondown}{b} \stackrel{\triangle}{=} \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{b}); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\overset{\leftharpoondown}{b}] \oplus\!\stackrel{\triangle}{=} \overset{\leftharpoondown}{x_4} \qquad \texttt{/*VIII*/} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ ([\,], \overset{\leftharpoondown}{b})\ \textbf{end}))\ \textbf{end}; \texttt{/*IX*/} \\
\qquad\qquad (\overleftarrow{x_2}, \overline{x_2}) \stackrel{\triangle}{=} \overleftarrow{x_1}\ \overleftarrow{a}; \\
\qquad\qquad (\overleftarrow{x_3}, \overline{x_3}) \stackrel{\triangle}{=} \overleftarrow{x_2}\ \overleftarrow{1} \\
\qquad \textbf{in}\ (\overleftarrow{x_3}, (\lambda \overset{\leftharpoondown}{x_3}\ \textbf{let}\ \overset{\leftharpoondown}{a} \stackrel{\triangle}{=} \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{a}); \\
\qquad\qquad\qquad\qquad\qquad \overset{\leftharpoondown}{x_1} \stackrel{\triangle}{=} \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_1}); \\
\qquad\qquad\qquad\qquad\qquad \overset{\leftharpoondown}{x_2} \stackrel{\triangle}{=} \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_2}); \\
\qquad\qquad\qquad\qquad\qquad \overset{\leftharpoondown}{1} \stackrel{\triangle}{=} \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{1}); \\
\qquad\qquad\qquad\qquad\qquad (\overset{\leftharpoondown}{x_2}, \overset{\leftharpoondown}{1}) \oplus\!\stackrel{\triangle}{=} \overline{x_3}\ \overset{\leftharpoondown}{x_3};\ \texttt{/*X*/} \\
\qquad\qquad\qquad\qquad\qquad (\overset{\leftharpoondown}{x_1}, \overset{\leftharpoondown}{a}) \oplus\!\stackrel{\triangle}{=} \overline{x_2}\ \overset{\leftharpoondown}{x_2};\ \texttt{/*XI*/} \\
\qquad\qquad\qquad\qquad\qquad [\,] \oplus\!\stackrel{\triangle}{=} \overset{\leftharpoondown}{x_1} \qquad\qquad \texttt{/*XII*/} \\
\qquad\qquad\qquad\qquad \textbf{in}\ ([\,], \overset{\leftharpoondown}{a})\ \textbf{end}))\ \textbf{end} \quad \texttt{/*XIII*/}
\end{array}
$$

To see how this works, we trace through the evaluation of this new backpropagator. First, $\overset{\leftharpoondown}{x_3}$ is bound to 1. Then, $\overset{\leftharpoondown}{a}$, $\overset{\leftharpoondown}{x_1}$, $\overset{\leftharpoondown}{x_2}$, and $\overset{\leftharpoondown}{1}$ are bound to zeros. Then, we apply $\overline{x_3}$ to 1. Since $\overline{x_3}$ is bound to $\lambda \overset{\leftharpoondown}{b} \ldots$, $\overset{\leftharpoondown}{b}$ is bound to 1 and $\overset{\leftharpoondown}{c}$ is bound to a zero. So far, the evaluation is the same as before. Now we see the first difference. The function $\lambda \overset{\leftharpoondown}{b} \ldots$ returns [1] paired with a zero, $\overset{\leftharpoondown}{x_2}$ is incremented by [1] to become [1], and $\overset{\leftharpoondown}{1}$ is incremented by a zero and remains a zero. Then, we apply $\overline{x_2}$ to [1]. Since $\overline{x_2}$ is bound to $\lambda \overset{\leftharpoondown}{x_4} \ldots$, $\overset{\leftharpoondown}{x_4}$ is bound to [1] and $\overset{\leftharpoondown}{b}$ is bound to a zero. Then $[\overset{\leftharpoondown}{b}]$ is incremented by [1]. This increments $\overset{\leftharpoondown}{b}$ by 1, allowing $\lambda \overset{\leftharpoondown}{x_4} \ldots$ to return [ ] paired with 1. The variable $\overset{\leftharpoondown}{x_1}$ is then incremented by [ ] and $\overset{\leftharpoondown}{a}$ 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 VII, VIII, IX, XII, and XIII 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 α-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 α-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 non-local 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[5] and a minimalist implementation of VLAD called STALIN∇ (pronounced Stalingrad).[6] VLAD and STALIN∇ support both forward-mode and reverse-mode AD, but in this paper we only describe reverse mode. VLAD and STALIN∇, 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 SCHEME data types supported are the empty list, Booleans, real numbers, pairs, and functions that take one argument and return one result. This is augmented with the machinery needed to support reverse-mode AD: reverse-tagged values and several novel primitive functions described below. This machinery augments the space of SCHEME values with reverse values, as described below.

[5]VLAD is an acronym for Functional Language for AD with a voiced F.

[6]The source code for STALIN∇ and all of the examples from this paper are available from `http://www.bcl.hamilton.ie/~qobi/stalingrad/software/toplas2006.tgz`.

—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 applications of the function CONS, as described below.

—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∇ accepts VLAD programs in SCHEME S-expression notation, in this paper we formulate VLAD programs in a more traditional mathematical notation that, *inter alia*, uses infix applications. While STALIN∇ 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. The distinction will be clear from context.

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 STALIN∇ 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. While STALIN∇ implements pairs and Booleans natively, for expository purposes, in this paper, we assume that pairs and Booleans are implemented using the following encoding:

$$\begin{array}{ll}
\textsc{Car}\; x & \triangleq\; x\; \lambda x_1\; \lambda x_2\; x_1\\
\textsc{Cdr}\; x & \triangleq\; x\; \lambda x_1\; \lambda x_2\; x_2\\
\textsc{Cons}\; x_1\; x_2\; x & \triangleq\; x\; x_1\; x_2\\
\mathbf{if}\; e_1\; \mathbf{then}\; e_2\; \mathbf{else}\; e_3\; \mathbf{fi} & \leadsto\; (e_1\; ((\lambda x\; e_2), (\lambda x\; e_3)))\; [\,]\;\text{ where } x \text{ is fresh}\\
e_1, e_2 & \leadsto\; \textsc{Cons}\; e_1\; e_2
\end{array}$$

where **true** and **false** are represented as CAR and CDR respectively. (The primitives must be aware of the representation of pairs and Booleans.) 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* $\overleftharpoon{x}$, 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 $\overleftarrow{\overleftarrow{x}}$. 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 order obeys a number of conditions: $\overleftarrow{x_1} \prec \overleftarrow{x_2}$, $x_1 \prec \overleftarrow{x_2}$, $\overleftarrow{x_1} \prec x_2$, $\overleftharpoon{x_1} \prec \overleftharpoon{x_2}$, $x_1 \prec \overleftharpoon{x_2}$, $\overleftharpoon{x_1} \prec x_2$, $\overline{x_1} \prec \overline{x_2}$,

$x_1 \prec \overline{x_2}$, and $\overline{x_1} \prec x_2$, if $x_1 \prec x_2$, and $x \prec \overleftharpoon{x} \prec \overleftharpoondown{x} \prec \overline{x}$. 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 that 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:

$$\mathbf{let}\ x_1 \triangleq e_1; \ldots; x_n \triangleq e_n\ \mathbf{in}\ x_n\ \mathbf{end}$$

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:

$$\begin{aligned}
&\mathbf{let}\ x_1 \triangleq e_1; x_2 \triangleq e_2; \ldots; x_n \triangleq e_n\ \mathbf{in}\ e\ \mathbf{end}\\
&\quad \rightsquigarrow\ \mathbf{let}\ x_1 \triangleq e_1\ \mathbf{in}\ \mathbf{let}\ x_2 \triangleq e_2; \ldots; x_n \triangleq e_n\ \mathbf{in}\ e\ \mathbf{end}\ \mathbf{end}
\end{aligned}$$

$$\mathbf{let}\ x_1 \triangleq e_1\ \mathbf{in}\ e\ \mathbf{end}\ \rightsquigarrow\ ((\lambda x_1\ e)\ e_1)$$

We further assume that all lambda expressions are α-converted.

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

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

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:

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

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

An *environment* σ is a finite map from variables to values. We use $\{\}$ to denote the empty environment and use σ_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 σ 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 $\mathbb{R} \rightarrow \mathbb{R}$, each language primitive b corresponds to some function $\mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$, each language primitive p corresponds to some unary predicate over values, and each language primitive q corresponds to some binary relation over pairs of values. We use t to denote any primitive and refer to closures and primitives collectively as *functions*.

We refer to a result of invoking $\overleftarrow{\mathcal{J}}$ (defined below) as a *reverse (transformed) value*. 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. While only non-closure values have explicit value tags, we can view transformed closure values as having implicit value tags corresponding to the tags on the argument variables of their lambda expression. This allows correct programs to obey a simple invariant: in the absence of sensitivity and backpropagator tags, the tag stack of a variable must be a prefix of the (possibly implicit) tag stack of a value to which that variable is bound.

We use (v_1, v_2) as shorthand for the *encoded 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 $[v_1, \ldots, v_l]$ to denote a list. It is shorthand for $(v_1, \ldots, v_l, [\,])$. To maintain the above invariant, we need to introduce transformed pairs and lists. We use $(v_1 ,_x v_2)$, $[\,]_x$, and $[v_1, \ldots, v_l]_x$ to denote pairs, empty lists, and lists that have been transformed according to the tag stack on the variable x:

$$\begin{array}{lcll}
(v_1 ,_x v_2) & \equiv & (v_1, v_2) & \text{when } x \in X\\
((\overleftarrow{\mathcal{J}}\ v_1) ,_{\overleftarrow{x}} (\overleftarrow{\mathcal{J}}\ v_2)) & \equiv & \overleftarrow{\mathcal{J}}\ (v_1 ,_x v_2) & \\
 & & & \\
[\,]_x & \equiv & [\,] & \text{when } x \in X\\
[\,]_{\overleftarrow{x}} & \equiv & \overleftarrow{[\,]_x} & \\
 & & & \\
[v_1, \ldots, v_l]_x & \equiv & (v_1 ,_x \cdots ,_x v_l ,_x [\,]_x) &
\end{array}$$

The implementation of $\overleftarrow{\mathcal{J}}$ will be defined below. As will be illustrated below, the transformation performed by $\overleftarrow{\mathcal{J}}$ is invertible. Furthermore, pair and list formation, both in the base case, as well as in the transformed case, are invertible. Thus we often use pair and list formation, as well as application of $\overleftarrow{\mathcal{J}}$, to indicate destructuring, 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. For our purposes, it suffices to take two expressions to be equivalent if they arise from the same program text, possibly via various transformations. This licenses approximating contextual equivalence, the notion that two expression evaluate to equal values for all environments, with a suitable decidable conservative approximation.

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

$$
\begin{array}{lll}
[\,] \oplus [\,] & \equiv [\,] & \\
r_1 \oplus r_2 & \equiv r_1 + r_2 & \\
\overleftarrow{v_1} \oplus \overleftarrow{v_2} & \equiv \overleftarrow{v_1 \oplus v_2} & \text{where } \overleftarrow{v_1} \text{ and } \overleftarrow{v_2} \text{ are reverse-tagged values} \\
t \oplus t & \equiv t & \\
(\sigma_1 \oplus \sigma_2)\ x & \equiv (\sigma_1\ x) \oplus (\sigma_2\ x) & \\
\langle \sigma_1, e\rangle \oplus \langle \sigma_2, e\rangle & \equiv \langle(\sigma_1 \oplus \sigma_2), e\rangle &
\end{array}
$$

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 $[v_1, \ldots, v_l]_x$ when $x'_1, \ldots, x'_l$ 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 $\mathbf{0}\ v$ to denote the zero that matches v:

$$
\begin{array}{ll}
\mathbf{0}\ [\,] & \equiv [\,] \\
\mathbf{0}\ r & \equiv 0 \\
\mathbf{0}\ \overleftarrow{v} & \equiv \overleftarrow{\mathbf{0}\ v} \\
 & \quad \text{where } \overleftarrow{v} \text{ is a reverse-tagged value} \\
\mathbf{0}\ t & \equiv [\,] \\
\mathbf{0}\ \langle \sigma, (\lambda x\ e)\rangle & \equiv [(\mathbf{0}\ (\sigma\ x'_1)), \ldots, (\mathbf{0}\ (\sigma\ x'_l))]_x \\
 & \quad \text{where } x'_1, \ldots, x'_l \text{ are the elements of} \\
 & \quad \mathcal{B}\ (\lambda x\ e) \text{ ordered by } \prec
\end{array}
$$

To define the reverse transform, we first define the following transformations on

let bindings:

$$
\begin{array}{lll}
\phi\{x_i \triangleq x_j\} & \equiv & \overleftarrow{x_i} \triangleq \overleftarrow{x_j}\\
\phi\{x_i \triangleq x_j\ x_k\} & \equiv & (\overleftarrow{x_i}, \overline{x_i}) \triangleq \overleftarrow{x_j}\ \overleftarrow{x_k}\\
\phi\{x_i \triangleq \lambda x\ e\} & \equiv & \overleftarrow{x_i} \triangleq \overleftarrow{\lambda x\ e}\\
\\
\rho\{x_i \triangleq x_j\} & \equiv & \overleftharpoon{x_j} \oplus\!\triangleq \overleftharpoon{x_i}\\
\rho\{x_i \triangleq x_j\ x_k\} & \equiv & (\overleftharpoon{x_j}, \overleftharpoon{x_k}) \oplus\!\triangleq \overline{x_j}\ \overleftharpoon{x_i}\\
\rho\{x_i \triangleq \lambda x\ e\} & \equiv & [\overleftharpoon{x'_1}, \ldots, \overleftharpoon{x'_l}]_x \oplus\!\triangleq \overleftharpoon{x_i} \ \text{ where } x'_1, \ldots, x'_l \text{ are the elements of}\\
 & & \qquad\qquad\qquad\qquad\quad \mathcal{B}\ (\lambda x\ e) \text{ ordered by } \prec
\end{array}
$$

We use ϕ to denote the forward-phase transformation and ρ to denote the reverse-phase transformation. Care must be taken in the second clause of the definition of ρ above to deal with the case where $j = k$. This corresponds to the case where $R_i = S_i$ in Section 2. In this case, both components of the pair returned by $\overline{x_j}\ \overleftharpoon{x_i}$ must be accumulated into the same variable.

Given these, the *reverse transform* $\overleftarrow{e}$ is:

$$
\begin{array}{ll}
\overleftarrow{\lambda x_0\ \textbf{let}\ x_1 \triangleq e_1;} \equiv & \lambda \overleftarrow{x_0}\ \textbf{let}\ \phi\{x_1 \triangleq e_1\};\\
\qquad\qquad \vdots & \qquad\qquad \vdots\\
\qquad\quad x_n \triangleq e_n & \qquad \phi\{x_n \triangleq e_n\}\\
\quad \textbf{in}\ x_n\ \textbf{end} & \textbf{in}\ (\overleftarrow{x_n}, (\lambda \overleftharpoon{x_n}\ \textbf{let}\ \overleftharpoon{x'_1} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x'_1});\\
 & \qquad\qquad\qquad\qquad \vdots\\
 & \qquad\qquad\qquad\quad \overleftharpoon{x'_l} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x'_l});\\
 & \qquad\qquad\qquad\quad \overleftharpoon{x_0} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_0});\\
 & \qquad\qquad\qquad\qquad \vdots\\
 & \qquad\qquad\qquad\quad \overleftharpoon{x_{n-1}} \triangleq \mathbf{0}\ (\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{x_{n-1}});\\
 & \qquad\qquad\qquad\quad \rho\{x_n \triangleq e_n\};\\
 & \qquad\qquad\qquad\qquad \vdots\\
 & \qquad\qquad\qquad\quad \rho\{x_1 \triangleq e_1\}\\
 & \qquad\qquad\quad \textbf{in}\ ([\overleftharpoon{x'_1}, \ldots, \overleftharpoon{x'_l}]_{x_0}, \overleftharpoon{x_0})\ \textbf{end}))\ \textbf{end}
\end{array}
$$

where $x'_1, \ldots, x'_l$ are the elements of $\mathcal{B}\ e$ ordered by $\prec$ and the reverse phase of $\overleftarrow{e}$ does not include any accumulation into sensitivity variables $\overleftharpoon{x}$ whenever $x \notin \{x_0\} \cup (\mathcal{B}\, e)$. The result of the above transformation is converted to A-normal form and then α-converted. Note that this transformation is invertible. This licenses the use of $\overleftarrow{\lambda x\ e}$ to denote argument destructuring in the definition of $\mathcal{B}$.

Using the above machinery, the primitives $\overleftarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}^{-1}$ can be implemented as follows:

$$
\begin{array}{lll}
\overleftarrow{\mathcal{J}}\ [\,] & \equiv \overleftarrow{[\,]} & \\
\overleftarrow{\mathcal{J}}\ r & \equiv \overleftarrow{r} & \\
\overleftarrow{\mathcal{J}}\ \overleftarrow{v} & \equiv \overleftarrow{\overleftarrow{v}} & \text{where } \overleftarrow{v} \text{ is a reverse-tagged value} \\
\overleftarrow{\mathcal{J}}\ t & \equiv \overleftarrow{t} & \\
\overleftarrow{\mathcal{J}}\ \langle \sigma, e\rangle & \equiv \langle \overleftarrow{\sigma}, \overleftarrow{e}\rangle & \text{where } \overleftarrow{\sigma}\ \overleftarrow{x} = \overleftarrow{\mathcal{J}}\ (\sigma\ x)\text{, for } x \in (\mathcal{F}\ e)\text{,} \\
 & & \text{and } \overleftarrow{\sigma}\ x = \sigma_0\ x\text{, for } x \in (\mathcal{F}\ \overleftarrow{e}) \setminus (\mathcal{F}\ e) \\
 & & \\
\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{v} & \equiv v & \text{where } \overleftarrow{v} \text{ is a reverse-tagged value} \\
\overleftarrow{\mathcal{J}}^{-1}\ \overleftarrow{t} & \equiv t & \\
\overleftarrow{\mathcal{J}}^{-1}\ \langle \overleftarrow{\sigma}, \overleftarrow{e}\rangle & \equiv \langle \sigma, e\rangle & \text{where } \sigma\ x = \overleftarrow{\mathcal{J}}^{-1}\ (\overleftarrow{\sigma}\ \overleftarrow{x})\text{, for } x \in \mathcal{F}\ e
\end{array}
$$

In the above, $\overleftarrow{t}$ denotes the transformation of the corresponding primitive t. These transformations will be defined below. Also note that $\overleftarrow{\mathcal{J}}^{-1}$ is the inverse of $\overleftarrow{\mathcal{J}}$. This licenses the use of $\overleftarrow{\mathcal{J}}$ to denote argument destructuring in the definition of $(v_1 {}_{,x}\, v_2)$, and also in VLAD expressions.

We use $\sigma[x \mapsto v]$ to denote the map that agrees with σ on all arguments except that it maps x to v. We have the following standard 'eval/apply' evaluator:

$$
\begin{array}{ll}
\mathcal{E}\ \sigma\ x & \equiv \sigma\ x \\
\mathcal{E}\ \sigma\ (e_1\ e_2) & \equiv \mathcal{A}\ (\mathcal{E}\ \sigma\ e_1)\ (\mathcal{E}\ \sigma\ e_1) \\
\mathcal{E}\ \sigma\ (\lambda x\ e) & \equiv \langle \sigma, (\lambda x\ e)\rangle \\
 & \\
\mathcal{A}\ u\ v & \equiv u\ v \\
\mathcal{A}\ b\ (v_1, v_2) & \equiv b\ v_1\ v_2 \\
\mathcal{A}\ p\ v & \equiv p\ v \\
\mathcal{A}\ q\ (v_1, v_2) & \equiv q\ v_1\ v_2 \\
\mathcal{A}\ \mathbf{0}\ v & \equiv \mathbf{0}\ v \\
\mathcal{A}\ \oplus\ (v_1, v_2) & \equiv v_1 \oplus v_2 \\
\mathcal{A}\ \overleftarrow{\mathcal{J}}\ v & \equiv \overleftarrow{\mathcal{J}}\ v \\
\mathcal{A}\ \overleftarrow{\mathcal{J}}^{-1}\ v & \equiv \overleftarrow{\mathcal{J}}^{-1}\ v \\
\mathcal{A}\ \langle \sigma, (\lambda x\ e)\rangle\ v & \equiv \mathcal{E}\ \sigma[x \mapsto v]\ e
\end{array}
$$

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:

$$
\begin{array}{l}
\overleftarrow{u} \equiv \mathcal{E}\ \sigma_0\ \lambda(\overleftarrow{\mathcal{J}}\ x)\ ((\overleftarrow{\mathcal{J}}\ (u\ x)), (\lambda \overline{y}\ ([\,], ((\mathcal{D}\ u\ x) \times \overline{y})))) \\
\overleftarrow{b} \equiv \mathcal{E}\ \sigma_0\ \lambda(\overleftarrow{\mathcal{J}}\ (x_1, x_2)) \\
\qquad\quad ((\overleftarrow{\mathcal{J}}\ (b\ (x_1, x_2))), \\
\qquad\quad\ (\lambda \overline{y}\ ([\,], (((\mathcal{D}_1\ b\ (x_1, x_2)) \times \overline{y}), ((\mathcal{D}_2\ b\ (x_1, x_2)) \times \overline{y}))))) \\
\overleftarrow{p} \equiv \mathcal{E}\ \sigma_0\ \lambda(\overleftarrow{\mathcal{J}}\ x)\ ((\overleftarrow{\mathcal{J}}\ (p\ x)), (\lambda \overline{y}\ ([\,], (\mathbf{0}\ x)))) \\
\overleftarrow{q} \equiv \mathcal{E}\ \sigma_0\ \lambda(\overleftarrow{\mathcal{J}}\ (x_1, x_2))\ ((\overleftarrow{\mathcal{J}}\ (q\ (x_1, x_2))), (\lambda \overline{y}\ ([\,], ((\mathbf{0}\ x_1), (\mathbf{0}\ x_2)))))
\end{array}
$$

In the above, $\mathcal{D}\ u\ x$ denotes a VLAD expression that evaluates the derivative of u at x and $\mathcal{D}_1\ b\ (x_1, x_2)$ and $\mathcal{D}_2\ b\ (x_1, x_2)$ denote VLAD expressions that evaluate the partial derivatives of 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:

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

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 of $\overleftarrow{\mathcal{J}}^{-1}$ nominally traverses its input. The commensurate increase in temporal complexity can be avoided by taking the implementations of the AD primitives $\mathbf{0}$, $\oplus$, $\overleftarrow{\mathcal{J}}$, and $\overleftarrow{\mathcal{J}}^{-1}$ to be lazy.

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

$$
\begin{aligned}
&\mathbf{let}\ x_1 \stackrel{\triangle}{=} (0, 0);\\
&\qquad x_2 \stackrel{\triangle}{=} (x_1, x_1);\\
&\qquad\ \vdots\\
&\qquad x_n \stackrel{\triangle}{=} (x_{n-1}, x_{n-1})\\
&\mathbf{in}\ x_n\ \mathbf{end}
\end{aligned}
$$

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 of the AD primitives $\mathbf{0}$, $\oplus$, $\overleftarrow{\mathcal{J}}$, and $\overleftarrow{\mathcal{J}}^{-1}$, when applied to nonscalar 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 constant factor, as can be verified be 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 $\overleftharpoon{x} = \overline{y}\ \overleftharpoon{y}$ 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∇ 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:

$$\nabla\ f\ x \stackrel{\triangle}{=} \textsc{Cdr}\ ((\textsc{Cdr}\ ((\overleftarrow{\mathcal{J}}\ f)\ (\overleftarrow{\mathcal{J}}\ x)))\ 1)$$

We then construct $\textsc{UnivariateArgmin}\ (f, \epsilon)$, 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:

$$\begin{array}{l}
\textsc{Argmin}\ (f, x_0, \epsilon) \stackrel{\triangle}{=} \\
\quad \textbf{let}\ g \stackrel{\triangle}{=} \nabla\ f\ x_0 \\
\quad \textbf{in if}\ ||g|| \leq \epsilon \\
\qquad \textbf{then}\ x_0 \\
\qquad \textbf{else}\ \textsc{Argmin} \\
\qquad\qquad (f, \\
\qquad\qquad\ (x_0 + (\textsc{UnivariateArgmin}\ ((\lambda k\ f\ (x_0 + k \times g)), \epsilon)) \times g), \\
\qquad\qquad\ \epsilon)\ \textbf{fi end}
\end{array}$$

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

$$\begin{array}{lcl}
\textsc{Argmax}\ (f, x_0, \epsilon) & \stackrel{\triangle}{=} & \textsc{Argmin}\ ((\lambda x\ -(f\ x)), x_0, \epsilon) \\
\textsc{Max}\ (f, x_0, \epsilon) & \stackrel{\triangle}{=} & f\ (\textsc{Argmax}\ (f, x_0, \epsilon))
\end{array}$$

Now let us construct a simple payoff function:

$$\textsc{Payoff}\ ([s, t], [u, v]) \stackrel{\triangle}{=} s^2 + t^2 - u^2 - v^2$$

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

$$\begin{array}{l}\textbf{let } x^* \stackrel{\triangle}{=} \textsc{Argmin}\ ((\lambda x\ \textsc{Max}\ ((\lambda y\ \textsc{Payoff}\ (x,y)), y_0, \epsilon)), x_0, \epsilon)\\ \textbf{in } (x^*, (\textsc{Argmax}\ ((\lambda y\ \textsc{Payoff}\ (x^*,y)), y_0, \epsilon)))\ \textbf{end}\end{array}$$

Note that finding x^* involves taking the derivative of $\lambda x\ \ldots$ which in turn, involves taking the second derivative of $\lambda y\ \ldots$. Also note that $\lambda y\ \ldots$ has a free reference to x, which is the argument to $\lambda x\ \ldots$. Finally note that $\lambda x\ \ldots$ calls Max, which calls Argmax, which calls Argmin, which calls ∇, which calls $\overleftarrow{\mathcal{J}}$. Since finding x^* involves taking the derivative of $\lambda x\ \ldots$, 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 as computing the original function [Christianson, 1992; Werbos, 1992, Section 10.7; 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:

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

where $\cdot$ denotes vector dot product. If we take $f\ [x,y] \stackrel{\triangle}{=} 2x^2 + 3xy + 4y^2$ 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 $\lambda x\ \ldots$ has a free reference to $\mathbf{v}$. Finally note that the outer $\overleftarrow{\mathcal{J}}$ transforms $\lambda x\ \ldots$, 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.

Reverse-mode AD entails postpending the reverse-phase computation to the forward-phase computation in reverse order. This can be performed on a single (leaf-node) function by a source-to-source transformation [Speelpenning, 1980]. It is difficult to properly perform the reverse-phase computation in the presence of function calls, because of the additional machinery necessary to save the intermediate forward-phase values and call trace. One common approach to doing this is to eschew source-to-source transformation in favor of recording a 'tape' of the forward-phase computation and transforming and replaying this tape to perform the reverse phase [Griewank et al., 1996].

TAPENADE [Hascoët and Pascual, 2004] extends the source-to-source transformation approach to first-order code that contains function calls by transforming each function in the original code into a pair of functions: one for the forward phase, and one for the reverse phase. The forward-phase function stores intermediate values on a tape for use during the reverse phase. Since the call graph of the original first-order code is static and determinable at compile time, the call graph of the reverse phase is also static. The tape need therefore only store the intermediate forward-phase values, and not the call trace. When applied to first-order code, the present method, with suitable standard compilation techniques, results in precisely the same transformation. Thus the present method is a strict generalization of the techniques used by TAPENADE to higher-order code in the presence of first-class AD operators. Furthermore, because it represents the tape as closures, the present method exposes that tape to standard compilation techniques even in the first-order case.

One derivative-taking method intended to perform reverse-mode AD in HASKELL is available [Karczmarczuk, 1998a, 2000a,b, 2001a]. In the absence of nesting, that method does calculate correct gradients. However it uses a computation graph different from that of reverse-mode AD. The method can be viewed as an implementation of forward-mode AD which calculates gradients by pairing each primal value in the n-dimensional input vector with a vector of n perturbation values, taking the perturbations of the i-th component of the input vector to be a vector of zeros with a single 1 in the i-th position. This would impose an overhead of $O(n)$ in time as compared with the original computation, while reverse-mode AD imposes $O(1)$ overhead. However the method does not construct the usual forward-mode AD computation graph, due to the way perturbations are represented: a real perturbation $\overrightarrow{v}$ is represented as $(\lambda z\ \overrightarrow{v} \times z)$. 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 operations need be defined:

$$(x:\mathbb{R})\ \tilde{\times}\ (y:\tilde{\mathbb{R}}) \stackrel{\triangle}{=} \lambda z\ y\ (x \times z) \qquad (x:\tilde{\mathbb{R}})\ \tilde{+}\ (y:\tilde{\mathbb{R}}) \stackrel{\triangle}{=} \lambda z\ (x\ z) + (y\ z)$$

These defer all arithmetic until an $\tilde{\mathbb{R}}$ number is applied to the $\mathbb{R}$ number 1. Arithmetic is overloaded to carry along vectors of these perturbations. Using the notation of Section 2:

$$\begin{aligned}
\overrightarrow{u}\ (x, \overrightarrow{\mathbf{x}}) &\stackrel{\triangle}{=} ((u\ x), (\textsc{Map}\ (\lambda \overrightarrow{x}\ (\mathcal{D}\ u\ x)\ \tilde{\times}\ \overrightarrow{x})\ \overrightarrow{\mathbf{x}}))\\
\overrightarrow{b}\ ((x_1, \overrightarrow{\mathbf{x}_1}), (x_2, \overrightarrow{\mathbf{x}_2})) &\stackrel{\triangle}{=} ((b\ (x_1, x_2)),\\
&\qquad (\textsc{Map2}\ (\lambda(\overrightarrow{x_1}, \overrightarrow{x_2})\ (\mathcal{D}_1\ b\ (x_1, x_2))\ \tilde{\times}\ \overrightarrow{x_1}\ \tilde{+}\\
&\qquad\qquad (\mathcal{D}_2\ b\ (x_1, x_2))\ \tilde{\times}\ \overrightarrow{x_2})\\
&\qquad\quad (\overrightarrow{\mathbf{x}_1}, \overrightarrow{\mathbf{x}_2})))
\end{aligned}$$

Fan-in of $\tilde{\mathbb{R}}$ values causes $\tilde{+}$ to be used, resulting in the same $\tilde{\mathbb{R}}$ number being called

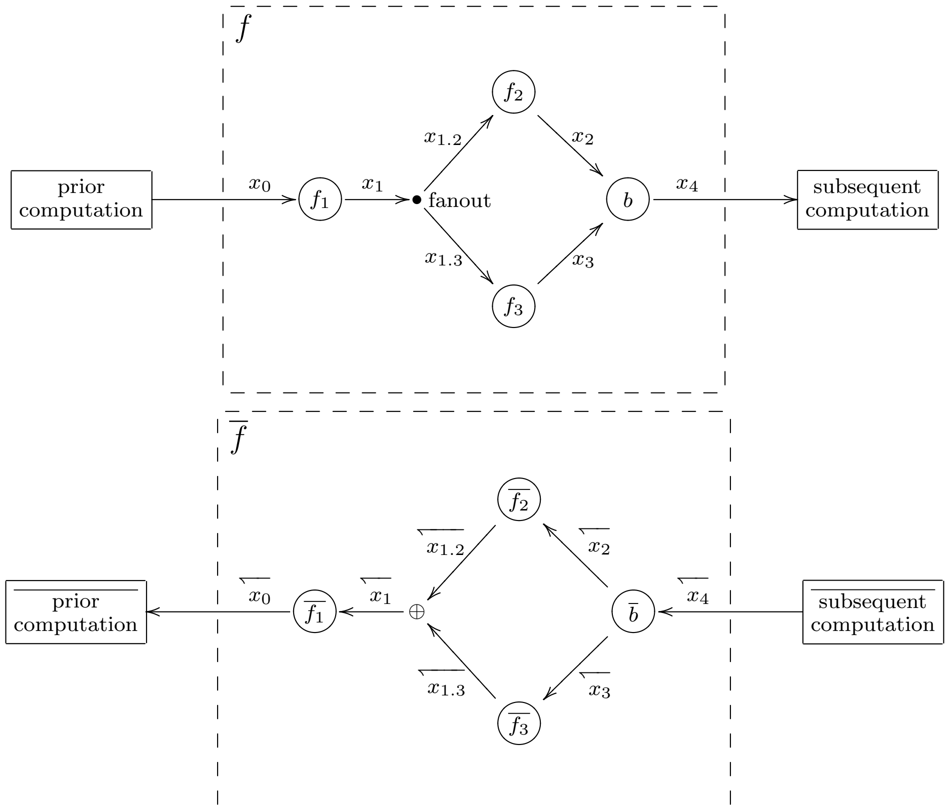


Fig. 5. With our method $\overline{f}$ (shown diagrammatically in the lower panel), the backpropagator for $f \triangleq \lambda x_0$ **let** $x_1 \triangleq f_1\ x_0$ **in** $b\ ((f_2\ x_1), (f_3\ x_1))$ **end** (shown diagrammatically in the upper panel), calls $\overline{f_2}$ and $\overline{f_3}$, the backpropagators for f_2 and f_3, with sensitivities $\overleftarrow{x_2}$ and $\overleftarrow{x_3}$, yielding $\overleftarrow{x_{1.2}} = \overline{f_2}\ \overleftarrow{x_2}$ and $\overleftarrow{x_{1.3}} = \overline{f_3}\ \overleftarrow{x_3}$, sums the results to yield $\overleftarrow{x_1} = \overleftarrow{x_{1.2}} \oplus \overleftarrow{x_{1.3}}$, the sensitivity of the output of f_1, where the addition in the reverse-phase $\overline{f}$ of the transformed $\overleftarrow{f}$ is needed because of the fanout of x_1 in the original f, and calculates the sensitivity $\overleftarrow{x_0} = \overline{f_1}\ \overleftarrow{x_1}$ of the input to f, thus calling $\overline{f_1}$ only once. Note that $(\overline{f_1}\ \overleftarrow{x_{1.2}}) \oplus (\overline{f_1}\ \overleftarrow{x_{1.3}}) = \overline{f_1}\ (\overleftarrow{x_{1.2}} \oplus \overleftarrow{x_{1.3}})$ because $\overline{f_1}$ is linear. (All backpropagators are linear as they are simply multiplication by the transpose of the Jacobian matrix.) Thus if $\overline{f}$ returned $(\overline{f_1}\ \overleftarrow{x_{1.2}}) \oplus (\overline{f_1}\ \overleftarrow{x_{1.3}})$ instead of $\overline{f_1}\ (\overleftarrow{x_{1.2}} \oplus \overleftarrow{x_{1.3}})$ it would be returning the correct value, but could entail additional computation. Fanout in the original computation causing addition in the reverse-phase computation is the key insight of reverse-mode AD. Here, the fanout of x_1 does not become apparent until b is invoked. The need to detect and properly transform fanout, in the presence of closures and with the possibility of multiple invocation of closures, is the reason for much of the machinery of the present method.

repeatedly with different arguments, which can increase the number of primitive arithmetic operations performed.

7. DISCUSSION

The primary technical difficulty we have solved is proper conversion of fanout in the primal to addition in the adjoint graph, in the face of closures and environments.

The need for this bookkeeping is shown diagrammatically in Figure 5. 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 encoded 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 run-time 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 AD exhibit 3 but not 1 and 2.

Our method involves a non-local 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 AD 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 problems 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.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software 22,* 2, 131–167.

HASCOËT, L. AND PASCUAL, V. 2004. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis.

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.

KARCZMARCZUK, J. 2000b. Lazy time reversal, and automatic differentiation.

KARCZMARCZUK, J. 2001a. Calcul des adjoints et programmation paresseuse.

KARCZMARCZUK, J. 2001b. Functional differentiation of computer programs. *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^5 report on the algorithmic language SCHEME. *Higher-Order and Symbolic Computation 11,* 1 (Sept.), 7–105.

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.

PEARLMUTTER, B. A. 1994. Fast exact multiplication by the Hessian. *Neural Computation 6,* 1, 147–160.

PEARLMUTTER, B. A. AND SISKIND, J. M. 2007. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 2007 Symposium on Principles of Programming Languages*. Nice, France, 155–60.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C*, 2nd 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. Tech. Rep. 99-190R, NEC Research Institute, Inc.

SISKIND, J. M. AND PEARLMUTTER, B. A. 2005. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In *Implementation and Application of Functional Languages—17th International Workshop, IFL'05*, A. Butterfield, Ed. Dublin, Ireland, 1–9. Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60.

SISKIND, J. M. AND PEARLMUTTER, B. A. 2007. Nesting forward-mode AD in a functional framework. *Higher Order and Symbolic Computation*. To appear.

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. *Comm. 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.

Received Month Year; revised Month Year; accepted Month Year