Efficient Implementation of a Higher-Order Language with Built-In AD*

Jeffrey Mark Siskind† Barak A. Pearlmutter‡

April 2016

We show that AD operators can be provided in a dynamic language without sacrificing numeric performance. To achieve this, general forward and reverse AD functions are added to a simple high-level dynamic language, and support for them is included in an aggressive optimizing compiler. Novel technical mechanisms are discussed, which have the ability to migrate the AD transformations from run-time to compile-time. The resulting system, although only a research prototype, exhibits startlingly good performance. In fact, despite the potential inefficiencies entailed by support of a functional-programming language and a first-class AD operator, performance is competitive with the fastest available preprocessor-based Fortran AD systems. On benchmarks involving nested use of the AD operators, it can even dramatically exceed their performance.

The Problem

Numerical programmers face a tradeoff. They can use a high-level language, like MATLAB or Python, that provides convenient access to mathematical abstractions like function optimization and differential equation solvers, or they can use a low-level language, like FORTRAN, to achieve high computational performance. The convenience of high-level languages results in part from the fact that they support many forms of run-time dependent computation: storage allocation and automatic reclamation, data structures whose size is run-time dependent, pointer indirection, closures, indirect function calls, tags and tag dispatching, *etc.*. This comes at a cost to the numerical programmer: the instruction stream contains a mix of floating-point instructions and instructions that form the scaffolding that supports run-time dependent computation. FORTRAN code, in contrast, achieves high floating-point performance by avoiding dilution of the instruction stream with such scaffolding.

This tradeoff is particularly poignant in the domain of automatic differentiation. Since the derivative *is* a higher-order function, it is most naturally incorporated into a language that supports higher-order functions in general. But on the other hand, efficiency of AD is often critical.

AD Implementation Strategies

One approach to AD involves a preprocessor performing a source-to-source transformation. In its simplest form, this can be viewed as translating a function:

```
double f(double x) {...}
```

into:

```
struct bundle {double primal; double tangent;};
struct bundle f_forward(struct bundle x) {...}
```

that, when passed a bundle of x and $\overline{x}$, returns a bundle of the primal value $f(x)$ and the tangent value $\overline{x}f'(x)$. When implemented properly, repeated application of this transformation can be used to produce variants of `f` that compute higher-order derivatives. Herein lies the inconvenience of this approach. Different optimizers might use derivatives of different order. Changing code to use a different optimizer would thus entail changing the build process to transform the objective function a different number of times. Moreover, the build process for nested application, such as multilevel optimization, would be tedious. One would need to transform the inner objective function, wrap it in a call to `optimize`, and then transform this resulting outer function.

A High-Performance Testbed Dynamic Language and Aggressive Compiler

We present a powerful and expressive formulation of forward and reverse AD based on a novel set of higher-order primitives, and develop the novel implementation techniques necessary to support highly efficient implementation of this formulation. We demonstrate that it is possible to combine the speed of FORTRAN with the expressiveness of a higher-level functional-programming language *augmented* with first-class AD.

*Extended abstract presented at the AD 2016 Conference, Sep 2016, Oxford UK.

†Corresponding Author, School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu

‡Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net

We exhibit a small but powerful language that provides a mechanism for defining a `derivative` operator that offers the convenience of the first approach with the efficiency of the second approach. This mechanism is formulated in terms of run-time reflection on the body of `f`, when computing `(derivative f)`, to transform it into something like `f_forward`. An optimizing compiler then uses whole-program inter-procedural flow analysis to eliminate such run-time reflection, as well as all other run-time scaffolding, yielding numerical code with FORTRAN-like (or super-FORTRAN) efficiency.

These results are achieved by combining (a) a novel formulation of forward and reverse AD in terms of a run-time reflexive mechanism that supports first-class nestable nonstandard interpretation with (b) the migration of the nonstandard interpretation to compile-time by whole-program inter-procedural flow analysis.

It should be noted that the implementation techniques invented for this purpose are, in principle, compatible with the optimizing compilers for procedural languages like Fortran. In other words, these techniques could be used to add in-language AD constructs to an aggressive optimizing Fortran or C compiler. In fact, a proof-of-concept has been exhibited which uses a small subset of these methods to build a Fortran AD pre-preprocessor which accepts a dialect of Fortran with in-language AD block constructs and which allows EXTERNAL FUNCTION arguments, and rewrites/expands the code, generating pure Fortran, and then uses existing tools like Tapenade to perform the required AD [1, 2].

Sketch of Implementation Technology

We present a novel approach that attains the advantages of both the overloading and transformation approaches. We define a novel functional-programming language, VLAD,[1] that contains mechanisms for transforming code into new code that computes derivatives.[2] These mechanisms apply to the source code that is, at least conceptually, part of closures, and such transformation happens, at least conceptually, at run time. Such transformation mechanisms replace the preprocessor, support a callee-derives programming style where the callee invokes the transformation mechanisms on closures provided by the caller, and allow the control flow of a program to determine the transformations needed to compute derivatives of the requisite order. Polyvariant flow analysis is then used to migrate the requisite transformations to compile time.[3]

We present a compiler that generates FORTRAN-like target code from a class of programs written in a higher-order functional-programming language with a first-class derivative operator. Our compiler uses whole-program inter-procedural flow analysis to drive a code generator. Our approach to flow analysis differs from that typically used when generating non-FORTRAN-like code. First, it is *polyvariant*. Monovariant flow analyses like 0-CFA [7] are unable to specialize higher-order functions. Polyvariant flow analysis is needed to do so. The need for polyvariant flow analysis is heightened in the presence of a higher-order derivative operator, *i.e.*, one that maps functions to their derivatives. Second, it is *union free*. The absence of unions in the abstract interpretation supports generation of code without tags and tag dispatching. The further absence of recursion in the abstract interpretation means that all aggregate data will have fixed size and shape that can be determined by flow analysis allowing the code generator to use unboxed representations without indirection in data access or run-time allocation and reclamation. The polyvariant analysis determines the target of all call sites allowing the code generator to use direct function calls exclusively. This, combined with aggressive inlining, results in inlined arithmetic operations, even when such operations are performed by (overloaded) function calls. The polyvariant analysis unrolls finite instances of what is written as recursive data structures. This, combined with aggressive unboxing, eliminates essentially all manipulation of aggregate data, including closures. Our limitation to union-free analyses and finite unrolling of recursive data structures is not as severe a limitation as it may seem. The main limitation relative to FORTRAN-like code is that we currently do not support arrays, though this restriction is easily lifted. Finally, the polyvariant analysis performs finite instances of reflection, migrating such reflective access to and creation of code from run time to compile time. This last aspect of our flow analysis is novel and crucial. Our novel AD primitives perform source-to-source transformation *within* the programming language rather than by a preprocessor, by reflective access to the code associated with closures and the creation of new closures with transformed code. Our flow analysis partially evaluates applications of the AD primitives, thus migrating such reflective access and transformation of code to compile time.

For those who are not compiler experts, an intuition for these techniques would be that hunks of code, including both entire procedures and smaller code blocks within procedures, are duplicated to make specialized versions for the different arguments that actually occur. For example, if an optimization routine `argmax` is called from two different places in the program using two different objective functions, two versions of `argmax` would be generated, each specialized to one of the objective functions. If `argmax` takes a gradient of the objective function it is passed, each of these two versions of `argmax` would have a known objective function, which would allow the AD transformation of that function to be migrated to compile time. Similar machinations allow much of the scaffolding around the numeric computation to be removed.

The effectiveness of these methods at attaining high floating point performance should be apparent from the benchmarking results shown in Figure 1.

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

[2] This differs from previous work on forward AD in a functional context [3, 4, 5, 6] which adopts an overloading approach.

[3] Existing transformation-based AD preprocessors, like ADIFOR and TAPENADE, use inter-procedural flow analysis for different incomparable purposes: not to eliminate run-time reflection, but to determine which subroutines to transform and which variables need tangents.

		particle				saddle				probabilistic-lambda-calculus		probabilistic-prolog		backprop		
		FF	FR	RF	RR	FF	FR	RF	RR	F	R	F	R	F	Fv	R
VLAD	STALIN∇	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	■	1.00
FORTRAN	ADIFOR	2.05	■	■	■	5.44	■	■	■	■	■	■	■	15.51	3.35	■
	TAPENADE	5.51	■	■	■	8.09	■	■	■	■	■	■	■	14.97	5.97	6.86
C	ADIC	■	■	■	■	■	■	■	■	■	■	■	■	22.75	5.61	■
C++	ADOL-C	■	■	■	■	■	■	■	■	■	■	■	■	12.16	5.79	32.77
	CPPAD	■	■	■	■	■	■	■	■	■	■	■	■	54.74	■	29.24
	FADBAD++	93.32	■	■	■	60.67	■	■	■	■	■	■	■	132.31	46.01	60.71
ML	MLTON	78.13	111.27	45.95	32.57	114.07	146.28	12.27	10.58	129.11	114.88	848.45	507.21	95.20	■	39.90
	OCAML	217.03	415.64	352.06	261.38	291.26	407.67	42.39	50.21	249.40	499.43	1260.83	1542.47	202.01	■	156.93
	SML/NJ	153.01	226.84	270.63	192.13	271.84	299.76	25.66	23.89	234.62	258.53	2505.59	1501.17	181.93	■	102.89
HASKELL	GHC	209.44	■	■	■	247.57	■	■	■	■	■	■	■	■	■	■
SCHEME	BIGLOO	627.78	855.70	275.63	187.39	1004.85	1076.73	105.24	89.23	983.12	1016.50	12832.92	7918.21	743.26	■	360.07
	CHICKEN	1453.06	2501.07	821.37	1360.00	2276.69	2964.02	225.73	252.87	2324.54	3040.44	44891.04	24634.44	1626.73	■	1125.24
	GAMBIT	578.94	879.39	356.47	260.98	958.73	1112.70	89.99	89.23	1033.46	1107.26	26077.48	14262.70	671.54	■	379.63
	IKARUS	266.54	386.21	158.63	116.85	424.75	527.57	41.27	42.34	497.48	517.89	8474.57	4845.10	279.59	■	165.16
	LARCENY	964.18	1308.68	360.68	272.96	1565.53	1508.39	126.44	112.82	1658.27	1606.44	25411.62	14386.61	1203.34	■	511.54
	MIT SCHEME	2025.23	3074.30	790.99	609.63	3501.21	3896.88	315.17	295.67	4130.88	3817.57	87772.39	49814.12	2446.33	■	1113.09
	MZC	1243.08	1944.00	740.31	557.45	2135.92	2434.05	194.49	187.53	2294.93	2346.13	57472.76	31784.38	1318.60	■	754.47
	MZSCHEME	1309.82	1926.77	712.97	555.28	2371.35	2690.64	224.61	219.29	2721.35	2625.21	60269.37	33135.06	1364.14	■	772.10
	SCHEME->C	582.20	743.00	270.83	208.38	910.19	913.66	82.93	69.87	811.37	803.22	10605.32	5935.56	597.67	■	280.93
	SCMUTILS	4462.83	■	■	■	7651.69	■	■	■	7699.14	■	83656.17	■	5889.26	■	■
	STALIN	364.08	547.73	399.39	295.00	543.68	690.64	63.96	52.93	956.47	1994.44	15048.42	16939.28	435.82	■	281.27

Figure 1: Comparative benchmark results for the `particle` and `saddle` examples [8], the `probabilistic-lambda-calculus` and `probabilistic-prolog` examples [9] and an implementation of backpropagation in neural networks using AD. Column labels are for AD modes and nesting: F for forward, Fv for forward-vector aka stacked tangents, RF for reverse-over-forward, etc. All run times normalized relative to a unit run time for STALIN∇ on the corresponding example except that run times for `backprop-Fv` are normalized relative to a unit run time for STALIN∇ on `backprop-F`. Pre-existing AD tools are named in blue, others are custom implementations. Key: ■ not implemented but could implement, including FORTRAN, C, and C++; ■ not implemented in pre-existing AD tool; ■ problematic to implement. All code available at http://www.bcl.hamilton.ie/~qobi/ad2016-benchmarks/.

Novelty and Significance

This paper makes two specific novel contributions:
(1) A novel set of higher-order functions (`j*`, `primal`, `tangent`, `bundle`, `zero`, `*j`) for performing forward and reverse AD in a functional language using source-to-source transformation via run-time reflection.
(2) A novel approach for using polyvariant flow analysis to eliminate such run-time reflection along with all other non-numerical scaffolding.

These are significant because they support AD with a programming style that is much more expressive and convenient than that provided by the existing preprocessor-based source-to-source transformation approach, yet still provides the performance advantages of that approach.

An alternative perspective would be that the language discussed here is in simple terms an eager lambda calculus augmented with a numeric basis and an AD basis. This is isomorphic to the intermediate forms used inside high-performance compilers, including aggressive optimizing FORTRAN compilers. As such, the techniques we discuss are useful not only for adding AD operators to functional programming languages, but also for adding AD support to compilers for imperative languages.

Acknowledgments

This work was supported, in part, by NSF grant 1522954-IIS and by Science Foundation Ireland grant 09/IN.1/I2637. 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.

References

[1] Alexey Radul, Barak A. Pearlmutter, and Jeffrey Mark Siskind. AD in Fortran, Part 1: Design. *arXiv:1203.1448*, 2012.

[2] Alexey Radul, Barak A. Pearlmutter, and Jeffrey Mark Siskind. AD in Fortran: Implementation via prepreprocessor. In *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, Fort Collins, Colorado, USA, July 23–27 2012. Springer. doi: 10.1007/978-3-642-30023-3_25. Also available as arxiv:1203.1450.

[3] Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14:35–57, 2001.

[4] Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy multivariate higher-order forward-mode AD. In *Proc of the 2007 Symposium on Principles of Programming Languages*, pages 155–60, Nice, France, January 2007. doi: 10.1145/1190215.1190242.

[5] Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–76, 2008. doi: 10.1007/s10990-008-9037-1.

[6] Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind. Confusion of tagged perturbations in forward automatic differentiation of higher-order functions. *Higher-Order and Symbolic Computation*, 2015. To appear. See also arXiv:1211.4892.

[7] Olin Grigsby Shivers, III. Control flow analysis in SCHEME. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–74, June 1988.

[8] Jeffrey Mark Siskind and Barak A. Pearlmutter. Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code. Technical Report TR-ECE-08-01, School of Electrical and Computer Engineering, Purdue University, January 2008. URL `http://docs.lib.purdue.edu/ecetr/367`.

[9] Jeffrey Mark Siskind. AD for probabilistic programming. NIPS 2008 workshop on Probabilistic Programming: Universal Languages and Inference; systems; and applications, 2008.