1-5-2008

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

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

Barak A. Pearlmutter
barak@cs.nuim.ie

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

Jeffrey Mark Siskind
School of Electrical and Computer Engineering
Purdue University, USA
qobi@purdue.edu

Barak A. Pearlmutter
Hamilton Institute
NUI Maynooth, Ireland
barak@cs.nuim.ie

Abstract

We exhibit an aggressive optimizing compiler for a functional-programming language which includes a first-class forward automatic differentiation (AD) operator. The compiler's performance is competitive with FORTRAN-based systems on our numerical examples, despite the potential inefficiencies entailed by support of a functional-programming language and a first-class AD operator. These results are achieved by combining (1) a novel formulation of forward AD in terms of a reflexive mechanism that supports first-class nestable nonstandard interpretation with (2) the migration to compile-time of the conceptually run-time nonstandard interpretation by whole-program inter-procedural flow analysis.

Categories and Subject Descriptors G.1.4 [*Quadrature and Numerical Differentiation*]: Automatic differentiation; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.4 [*Processors*]: Code generation, Compilers, Optimization; F.3.2 [*Semantics of Programming Languages*]: Partial evaluation, Program analysis

General Terms Performance, Languages

Keywords Forward AD, Source-to-source transformation, Reflection, Higher-order functions, Nonstandard interpretation

1. Introduction

Numerical computing could greatly benefit from more expressive programming languages, as evidenced by growing use of MATLAB, OCAML, SCIPY, HASKELL, etc. The impact of functional programming languages on the numerical programming community has, however, been seriously diluted by two issues. First, the most important higher-order function for numerical programming, namely an exact efficient derivative operator, is not available; and second, the speed penalty has been dramatic. In this paper 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 automatic differentiation (AD).

Consider some stereotypical numerical code and its associated execution model. Numerical code typically does not use union types and thus its execution model does not use tags or tag dispatching. In numerical code, all aggregate data typically has fixed size and shape that can be determined at compile time. Thus in the execution model, such aggregate data is unboxed and does not require indirection for data access and run-time allocation and reclamation. Numerical code typically does not use higher-order functions. Thus in the execution model, all function calls are to known targets and do not involve indirection or closures. Numerical code is typically written in languages that do not support reflection so code cannot be accessed, modified or created during execution. We refer to such code and its corresponding execution model as FORTRAN-like. When properly compiled, FORTRAN-like numerical code can exhibit significantly greater performance than numerical code written in a non-FORTRAN-like style compiled with typical compilers.

This performance comes at a price in expressiveness, so numerical programmers face a tradeoff. They can use a high-level language, like MATLAB, 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 tradeoff is particularly poignant in the domain of *automatic differentiation* or AD. AD is a collection of techniques for evaluating the derivative of a function specified by a computer program at a particular input.[1] In its simplest form, AD could be provided with a simple API:

`(derivative` $f : \mathbb{R} \to \mathbb{R}$ $x : \mathbb{R}) : \mathbb{R}$

or a curried variant (Sussman et al. 2001). This formulation as a higher-order function would allow construction of a whole hierarchy of mathematical concepts, like partial derivatives, gradients, function optimization, differential-equation solvers, etc., that are built upon the notion of a derivative. Moreover, once one defines such abstractions, it is natural and useful to nest them:

```
(optimize (λ (x) (optimize (λ (y) ...) ...)) ...)
(optimize (λ (x) (solve-ode (λ (y) ...) ...)) ...)
```

[1] AD is distinct from symbolic manipulation as performed by computer algebra systems, like MACSYMA and MAPLE, which cannot handle the control flow and aggregate data structures that distinguish *programs* from mathematical expressions and thus cannot handle the examples in Section 7.

Since the derivative *is* a higher-order function, it is most naturally incorporated into a language that supports higher-order functions in general. We present a powerful and expressive formulation of forward 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.

2. Review of Forward AD

A *dual number* (Clifford 1873) is a pair $\langle x, \overline{x} \rangle$ of two real numbers x and $\overline{x}$, which supports arithmetic by regarding it as a truncated power series $x + \overline{x}\varepsilon$. This arithmetic can be derived be analogy with arithmetic on complex numbers, where instead of taking $\mathrm{i}^2 = -1$ we take $\varepsilon^2 = 0$ but $\varepsilon \neq 0$. Like complex numbers, dual numbers are represented as simple tuples rather than as terms, but also like complex numbers we write them using the notation $x + \overline{x}\varepsilon$ for ease of manipulation. Dual numbers can also be viewed as tangent-bundle pairs, and for this reason we refer to x and $\overline{x}$ as the primal and tangent values of $x + \overline{x}\varepsilon$.

Forward AD[2] computes the derivative of a univariate function $f : \mathbb{R} \to \mathbb{R}$ at a scalar point c by evaluating $f\ (c + \varepsilon)$ under a nonstandard interpretation replacing real numbers with dual numbers and extracting the coefficient of ε in the result. To see how this works, let us manually apply the mechanism to a simple example: computing the first derivative of $f(x) = x^4 + 2x^3$ at $x = 3$. To do this, we first evaluate $f(3 + \varepsilon)$:

$$\begin{aligned} f(3+\varepsilon) &= (3+\varepsilon)^4 + 2(3+\varepsilon)^3 \\ &= (81 + 108\varepsilon) + 2(27 + 27\varepsilon) \\ &= 135 + 162\varepsilon \end{aligned}$$

From this we can extract the tangent, which gives the derivative, 162. Note that the above makes use of the restriction that $\varepsilon^2 = 0$, dropping the ε^2, ε^3, and ε^4 terms when evaluating the expressions $(3+\varepsilon)^3 = 27+27\varepsilon$ and $(3+\varepsilon)^4 = 81+108\varepsilon$. This is the essence of traditional forward AD when limited to the univariate case.

AD, in both its forward (Wengert 1964) and reverse (Speelpenning 1980) variants, is widely used for scientific and engineering computation. See `www.autodiff.org` for a plethora of implementations of forward and reverse AD in a multitude of programming languages. Broadly speaking, these implementations employ one of two general approaches for performing the nonstandard interpretation, described above, that is characteristic of forward AD. One is to represent the dual numbers $\langle x, \overline{x} \rangle$ (henceforth referred to simply as bundles, as a contraction of tangent-bundle pair) as objects and overload the arithmetic primitives to manipulate such objects. FADBAD++ (Bendtsen and Stauning 1996) is one of example of this approach. The other is to transform the source code, replacing each real variable x with a pair of real variables x and $\overline{x}$ and augmenting the source code with expressions and statements to compute the $\overline{x}$ values. ADIFOR (Bischof et al. 1996) and TAPENADE (Hascoët and Pascual 2004) are two examples of this approach.

These two approaches exhibit complementary tradeoffs. The overloading approach, particularly when it allows arithmetic operations to apply to either real numbers or bundles, supports a *callee derives* programming style. A function optimizer can be be written as a higher-order function, taking an objective function as its argument. The optimizer can invoke the objective function with a bundle to compute its derivative and perform gradient-based optimization, without knowledge of the caller. In contrast, the transformation approach requires a *caller derives* programming style. The optimizer takes two function arguments, the objective function and its derivative, and the caller must arrange for the build system to transform the code for the objective function into code for its derivative. The overloading approach thus supports a greater level of modularity while the transformation approach requires exposing the need for derivatives in the signatures of functions.

The overloading approach exhibits another advantage. When implemented correctly, one can take derivatives of functions that in turn take derivatives of other functions. (The utility of doing so is illustrated in Section 7.) This involves computing higher-order derivatives. Some overloading-based implementations can compute derivatives of arbitrary order, even when the requisite order is not explicitly specified and only implicit in the control-flow of the program. When implemented correctly, the transformation approach can also transform transformed code to compute higher-order derivatives. The difference is that, since the transformation is typically done by a preprocessor, the preprocessor must be explicitly told which higher-order derivatives are needed.

In contrast, the overloading approach exhibits a computational cost that is not exhibited by the transformation approach. Unless specifically optimized, bundles must be allocated at run time, accessing the components of bundles requires indirection, and overloaded arithmetic can require run-time dispatch and perhaps even indirect function calls. The transformation approach, however, can yield FORTRAN-like code without these run-time costs and has thus become the method of choice in communities where the speed of numerical code is of paramount importance.

We present a novel approach that attains the advantages of both the overloading and transformation approaches. We define a novel functional-programming language, VLAD, that contains mechanisms for transforming code into new code that computes derivatives. 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]

3. Overview

Given the formulation from the previous section, evaluation of `(`f x`)` under the nonstandard interpretation implied by forward AD requires two things. First, one must transform f so that it operates on bundles instead of reals. We introduce the function `j*` to accomplish this. Second, one must bundle x with a tangent. We introduce the function `bundle` to accomplish this.

When computing simple derivatives, the tangent of the independent variable is one. This is accomplished by evaluating the expression `((j*` f`)` `(bundle` x `1))`. This yields a bundle containing the value of $f(x)$ with its derivative $f'(x)$. We introduce the functions `primal` and `tangent` to extract these components. With these, the derivative operator for functions $\mathbb{R} \to \mathbb{R}$ can be formulated as a higher-order function:

```
(define ((derivative f) x)
 (tangent ((j* f) (bundle x 1))))
```

[2] There is a complementary variant of AD called reverse AD (Speelpenning 1980). While the programming language that we present also supports reverse AD, our focus here is on forward AD and the compiler optimizations that we have implemented to support forward AD. Our intent is to extend these compiler optimizations to support reverse AD in the future.

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

Several complications arise. The function f may call other functions, directly or indirectly, and all of these may call primitives. All of these need to be transformed. We assume that primitives are not inlined (at least conceptually) and that every function or primitive that is called is reachable as a (possibly nested) value of a free variable closed over by f. A reflective mechanism called `map-closure` (Siskind and Pearlmutter 2007) is used to access these values from otherwise opaque closures.

We assume that all constants accessed by f are represented as values of free variables closed over by f (i.e., constant conversion). These, along with other closed-over variables (that are treated as constants for the purpose of computing derivatives) must have all (potentially nested) reals bundled with zero. Thus `j*` conceptually incorporates the mechanism of `bundle`.

Similarly, the input data x might be aggregate. In this case, partial derivatives can be computed by taking one real component to be the independent variable, and thus bundled with one, and the other real components to be constants, and thus bundled with zero. Alternatively, directional derivatives can be computed by bundling the real components of x with the corresponding components of the direction vector. Thus we generalize `bundle` to take aggregate data paired with an aggregate tangent containing the direction-vector components. It is necessary to have the primal and tangent arguments to bundle have the same shape. Thus when the primal argument contains discrete values, we fill the corresponding components of the tangent argument with the same values as the primal argument (see Section 4.1).

Just as the input data might be aggregate, the result of a function might also be aggregate. Accordingly, we generalize `primal` and `tangent` to take arbitrary aggregate data that contains (possibly nested) bundles as arguments and traverse such data to yield result data of the same shape containing only the primal or tangent components of these (possibly nested) bundles. Such aggregate data may contain opaque closures. So that `primal` and `tangent` can traverse these closures, they too are formulated with using the `map-closure` reflective mechanism.

The aggregate value x may contain closures (which get called by f). Thus these (and all functions and closed-over reals that they can access) also need to be transformed. Thus `bundle` conceptually incorporates the mechanism of `j*`. The mechanism of `j*` conceptually is the same as bundling all closed-over values with zero. However, since closed-over values are often opaque closures, we need a way to construct an appropriate closure as a tangent value whose slots are zero. We introduce the `zero` function to map an arbitrary data structure x, possibly containing possibly nested closures, to a tangent value of the same shape with zero tangent values in all slots that correspond to those in x that contain reals.

With this, `j*` can be defined as:

```
(define (j* x) (bundle x (zero x)))
```

so long as `bundle` transforms primitives and `primal` and `tangent` are able to transform transformed primitives back to the corresponding original primitives.

4. VLAD: A Functional Language for AD

VLAD is a simple higher-order functional-programming language designed to support AD. It resembles SCHEME (Clinger and Rees 1991), differing in the following respects:

- The only SCHEME datatypes supported are the empty list, booleans, reals, pairs, and procedures.
- Only a subset of the builtin SCHEME procedures and syntax are supported.
- Rest arguments are not supported.
- The construct `cons` is builtin syntax.
- The construct `list` is a macro:

  $$\texttt{(list)} \quad \rightsquigarrow \quad \texttt{'()}$$
  $$\texttt{(list } e_1\ e_2 \ldots\texttt{)} \quad \rightsquigarrow \quad \texttt{(cons } e_1\ \texttt{(list } e_2 \ldots\texttt{))}$$

- Procedure parameters p can be variables, `'()` to indicate an argument that is ignored, or `(cons` p_1 p_2`)` to indicate the appropriate destructuring.
- All procedures take exactly one argument and return exactly one result. This is accomplished in part by the basis, in part by the following transformations:

  $$\texttt{(}e_1\texttt{)} \quad \rightsquigarrow \quad \texttt{(}e_1\ \texttt{'())}$$
  $$\texttt{(}e_1\ e_2\ e_3\ e_4 \ldots\texttt{)} \quad \rightsquigarrow \quad \texttt{(}e_1\ \texttt{(cons* } e_2\ e_3\ e_4 \ldots\texttt{))}$$
  $$\texttt{(lambda () } e\texttt{)} \quad \rightsquigarrow \quad \texttt{(lambda ((cons*)) } e\texttt{)}$$
  $$\texttt{(lambda (}p_1\ p_2\ p_3 \ldots\texttt{) } e\texttt{)} \rightsquigarrow \texttt{(lambda ((cons* } p_1\ p_2\ p_3 \ldots\texttt{)) } e\texttt{)}$$

  together with a `cons*` macro:

  $$\texttt{(cons*)} \quad \rightsquigarrow \quad \texttt{'()}$$
  $$\texttt{(cons* } e_1\texttt{)} \quad \rightsquigarrow \quad e_1$$
  $$\texttt{(cons* } e_1\ e_2\ e_3 \ldots\texttt{)} \quad \rightsquigarrow \quad \texttt{(cons } e_1\ \texttt{(cons* } e_2\ e_3 \ldots\texttt{))}$$

  and by allowing `list` and `cons*` as parameters.

The above, together with the standard SCHEME macro expansions, a macro for `if`:

$$\texttt{(if } e_1\ e_2\ e_3\texttt{)} \rightsquigarrow \texttt{(if-procedure } e_1\ \texttt{(lambda () } e_2\texttt{) (lambda () } e_3\texttt{))}$$

and conversion of constants into references to variables bound in a top-level basis environment (i.e., constant conversion) suffice to transform any program into the following core language:

$$e ::= x \mid \texttt{(lambda (}x\texttt{) } e\texttt{)} \mid \texttt{(}e_1\ e_2\texttt{)} \mid \texttt{(letrec ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) } e\texttt{)} \mid \texttt{(cons } e_1\ e_2\texttt{)}$$

We often abbreviate `(lambda (`x`) `e`)` and `(cons` e_1 e_2`)` as $(\lambda x\ e)$ and (e_1, e_2) respectively. For expository purposes, we omit discussion of `letrec` for the remainder of this paper.

We use x to denote variables, e to denote expressions, and v to denote VLAD values. Values are either scalar or aggregate. Scalars are either discrete, such as the empty list, booleans, or primitive procedures (henceforth primitives), or continuous, i.e., reals. Aggregate values are either closures $\langle \sigma, e \rangle$ or pairs (v_1, v_2), where σ is an environment, a (partial) map from variables to values, represented extensionally as a set of bindings $x \mapsto v$. Pairs are constructed by the core syntax (e_1, e_2) and the components of pairs can be accessed by the primitives `car` and `cdr`.

4.1 The Forward AD Basis

We augment the space of aggregate values to include *bundles* denoted as $(v_1 \rhd v_2)$. We refer to the first component of a bundle as the *primal* value and the second component of a bundle as the *tangent* value. Unlike pairs, which can contain arbitrary values as components, bundles are constrained so that the tangent is a member of the tangent space of the primal. We will define the tangent space momentarily. We augment the basis with the primitive `bundle` to construct bundles, the primitives `primal` and `tangent` to access the components of bundles, and the primitive `zero` to construct zero elements of the tangent spaces. Part of the complexity of the following formulation is due to the fact that these notions are generalized to support aggregate datatypes, not just reals, and also nested and composed application of `j*`.

We denote an element of the tangent space of a value v as $\overline{v'}$ and an element of the bundle space of a value v, i.e., the space of bundles $(v \rhd \overline{v'})$, as $\overrightarrow{v}$. We will formally define the tangent and bundle spaces momentarily. We first give the informal intuition.

Defining the tangent and bundle spaces for reals is straightforward. The tangent of a real is a real and the bundle of a real with its

real tangent is a pair thereof. We use $(v_1 \rhd v_2)$ instead of (v_1, v_2) to distinguish bundles from pairs created by `cons`. The definition of tangent and bundle spaces becomes more involved for other types of data. Conceptually, at least, we can take the bundle space of any value v_1 to be the space of bundles $(v_1 \rhd v_2)$ where v_2 is a member of an appropriate tangent space of v_1. For now, let us take the tangent of a pair to also be a pair. (We will justify this momentarily.) With this, we can take the bundle space of a pair (v_1, v_2) to be $((v_1, v_2) \rhd (v_3, v_4))$. Alternatively, we can interleave the components of the tangent with the components of the primal: $((v_1 \rhd v_3), (v_2 \rhd v_4))$ The former has the advantage that extracting the primal and tangent is simple but the disadvantage that extracting the `car` and `cdr` requires traversing the data structure. The latter has complementary tradeoffs.

Conceptually, at least, we can use either representation for the bundle space of closures. However, the interleaved representation has an advantage in that it is also a closure:

$$\langle \{x_1 \mapsto (v_1 \rhd v_1'), \ldots, x_n \mapsto (v_n \rhd v_n')\}, e \rangle$$

and thus can be invoked by the same evaluation mechanism as ordinary closures for primal values. The non-interleaved representation, however, is not a closure:

$$(\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, e \rangle \rhd (\ldots, v_1', \ldots, v_n', \ldots))$$

It is a primal closure bundled with an element of the tangent space of that closure, whatever that is, and would require a novel evaluation mechanism. This motivates using the interleaved representation, at least for closures.

Conceptually, at least, the above issue affects only closures. One could adopt either representation for other aggregate data. However, we wish our programs to exhibit another desirable property. In the absence of AD, the semantics of a program is unchanged when one replaces a builtin aggregate datatype, like pairs, with an encoding as closures, like that of Church (1941) or Scott. This implies, that conceptually at least, all aggregate data must use the interleaved representation.

This creates an ambiguity: does $((v_1 \rhd v_3), (v_2 \rhd v_4))$ represent a pair of two bundles $(v_1 \rhd v_3)$ and $(v_2 \rhd v_4)$ or a bundle of two pairs (v_1, v_2) and (v_3, v_4) that have been interleaved? To resolve this ambiguity, we introduce the notion of a 'bundled' pair $(v_1 \overrightarrow{,} v_2)$. We augment our core syntax with expressions $(e_1 \overrightarrow{,} e_2)$ to construct bundled pairs. Note that we must support the ability to represent and construct multiply bundled pairs $(v_1 \overrightarrow{\overrightarrow{,}} v_2)$, $(v_1 \overrightarrow{\overrightarrow{\overrightarrow{,}}} v_2)$, etc.

A similar ambiguity arises for closures, i.e., does:

$$\langle \{x_1 \mapsto (v_1 \rhd v_1'), \ldots, x_n \mapsto (v_n \rhd v_n')\}, (\lambda x\ e) \rangle$$

represent a primal closure that happens to close over bundle values or a bundled closure? To resolve this ambiguity, we adopt a tagging scheme $\overrightarrow{x}$ for variables x to indicate that they contain bundles. The above would indicate a primal closure (that happens to close over bundle values) while:

$$\langle \{\overrightarrow{x_1} \mapsto (v_1 \rhd v_1'), \ldots, \overrightarrow{x_n} \mapsto (v_n \rhd v_n')\}, (\lambda\, \overrightarrow{x}\ \overrightarrow{e}) \rangle$$

would indicate a bundled closure. We transform the bodies e of the lambda expressions associated with closures to access the suitably tagged variables and also to construct suitably bundled pairs.

The question then arises: what form should the tangent space of aggregate data take? The tangent of a piece of aggregate data must contain the same number of reals as the corresponding primal. Conceptually, at least, one might consider representing the tangent of one object with an object of a different type or shape, e.g., taking the tangent of a closure to be constructed out of pairs. However, one can show that any function f that only rearranges a data structure containing reals to a different data structure containing reals, without performing any operations on such reals, must exhibit the following property:

$$\texttt{((j* } f\texttt{) x)} = \texttt{(bundle (}f\texttt{ (primal x)) (}f\texttt{ (tangent x)))}$$

Since f must perform the same rearrangement on both the primal and the tangent, it must be insensitive to its type or shape. As VLAD functions can be sensitive to their argument's type or shape, this implies that the tangent of an aggregate object must be of the same type and shape as the corresponding primal. This further implies that the tangent of a discrete object such as the empty list, a boolean, or a primitive must be the same as that object.

We now formalize the above intuition. We introduce a mechanism for creating a new variable $\overrightarrow{x}$ that corresponds to an existing variable x (which may itself be such a newly created variable). The variable $\overrightarrow{x}$ must be distinct from any existing variable including x. Any variable $\overrightarrow{x}$ will contain an element of the bundle space of the corresponding variable x. Our AD transformations rely on a bijection between the space of variables x and the space of variables $\overrightarrow{x}$.

We introduce the following transformation between the space of expressions e that manipulate primal values to the space of expressions $\overrightarrow{e}$ that manipulate bundle values:

$$\begin{array}{rcl}
\overrightarrow{(\lambda x\ e)} & \leadsto & (\lambda\, \overrightarrow{x}\ \overrightarrow{e}) \\
\overrightarrow{(e_1\ e_2)} & \leadsto & (\overrightarrow{e_1}\ \overrightarrow{e_2}) \\
\overrightarrow{(e_1, e_2)} & \leadsto & (\overrightarrow{e_1} \overrightarrow{,} \overrightarrow{e_2})
\end{array}$$

We require this to be a bijection since `bundle` will map e to $\overrightarrow{e}$ and `primal` and `tangent` will map $\overrightarrow{e}$ back to e. Note that the code $\overrightarrow{e}$ is largely the same as the code e except for two differences. First, the variable binders and accesses have been mapped from x to $\overrightarrow{x}$. This is simply α conversion. Second, the `cons` expressions (e_1, e_2) are mapped to $(\overrightarrow{e_1} \overrightarrow{,} \overrightarrow{e_2})$ where '$\overrightarrow{,}$' denotes a new kind of expression that constructs bundled pairs.

We now can formally define the tangent space of VLAD values:

$$\begin{array}{l}
\overline{v'} = v \quad \text{when } v \text{ is a discrete scalar} \\
\overline{v'} \in \mathbb{R} \quad \text{when } v \in \mathbb{R} \\
\overrightarrow{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, (\lambda x\ e) \rangle} = \\
\quad \langle \{x_1 \mapsto \overline{v_1'}, \ldots, x_n \mapsto \overline{v_n'}\}, (\lambda x\ e) \rangle \\
\overrightarrow{(v \rhd \overline{v'})} = (\overrightarrow{v'} \rhd \overrightarrow{\overline{v'}}) \\
\overrightarrow{(v_1, v_2)} = (\overrightarrow{v_1'}, \overrightarrow{v_2'})
\end{array}$$

and the corresponding bundle space:

$$\overrightarrow{v} = (v \rhd \overrightarrow{v'}) \quad \text{when } v \text{ is a non-primitive scalar}$$

$$\overrightarrow{v} = \langle \sigma, (\lambda\, \overrightarrow{x}\ (\texttt{bundle}\ ((v\ (\texttt{primal}\ \overrightarrow{x})),\ (*\ ((v^{(1)}\ (\texttt{primal}\ \overrightarrow{x})),\ (\texttt{tangent}\ \overrightarrow{x}))))))\rangle$$
when v is a primitive $\mathbb{R} \to \mathbb{R}$

$$\overrightarrow{v} = \langle \sigma, (\lambda\, \overrightarrow{x}\ (\texttt{bundle}\ ((v\ (\texttt{primal}\ \overrightarrow{x})),\ (+\ ((*\ ((v^{(1,0)}\ (\texttt{primal}\ \overrightarrow{x})),\ (\texttt{car}\ (\texttt{tangent}\ \overrightarrow{x})))),\ (*\ ((v^{(0,1)}\ (\texttt{primal}\ \overrightarrow{x})),\ (\texttt{cdr}\ (\texttt{tangent}\ \overrightarrow{x}))))))))))\rangle$$
when v is a primitive $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$

$$\overrightarrow{v} = \langle \sigma, (\lambda\, \overrightarrow{x}\ (\texttt{j*}\ (v\ (\texttt{primal}\ \overrightarrow{x}))))\rangle$$
when v is a primitive predicate

$$\overrightarrow{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, (\lambda x\ e)\rangle} = \langle \{\overrightarrow{x_1} \mapsto \overrightarrow{v_1}, \ldots, \overrightarrow{x_n} \mapsto \overrightarrow{v_n}\}, \overrightarrow{(\lambda x\ e)}\rangle$$

$$\overrightarrow{(v \rhd \overrightarrow{v'})} = (\overrightarrow{v} \rhd \overrightarrow{\overrightarrow{v'}})$$

$$\overrightarrow{(v_1, v_2)} = (\overrightarrow{v_1}, \overrightarrow{v_2})$$

In the above, $v^{(1)}$ denotes the derivative of v, and $v^{(1,0)}$ and $v^{(0,1)}$ denote the partial derivatives of v with respect to its first and second arguments. A finite number of such explicit derivatives are needed for the finite set of primitives. We only show how to transform arithmetic primitives. Transformations of other primitives, such as `if-procedure`, `car`, and `cdr`, as well as the AD primitives `bundle`, `primal`, `tangent`, and `zero` themselves, follow from the earlier observation about functions that only rearrange aggregate data. Also note that the environment σ in the closures created for transformed primitives must map all of the free variables to their values in the top-level environment. This includes v itself, as well as `primal`, `tangent`, `bundle`, `j*`, `car`, `cdr`, `+`, `*`, and anything needed to implement $v^{(1)}$, $v^{(1,0)}$, and $v^{(0,1)}$.

We now can give formal definitions of the AD primitives. The primitive `bundle` is defined as follows:

$$\texttt{bundle}\ (v, \overrightarrow{v'}) \stackrel{\triangle}{=} (v \rhd \overrightarrow{v'}) \quad \text{when } v \text{ is a non-primitive scalar}$$

$$\texttt{bundle}\ (v, \overrightarrow{v'}) \stackrel{\triangle}{=} \overrightarrow{v} \quad \text{when } v \text{ is a primitive}$$

$$\texttt{bundle}\ (\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, (\lambda x\ e)\rangle, \overrightarrow{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, (\lambda x\ e)\rangle}) \stackrel{\triangle}{=} \langle \{\overrightarrow{x_1} \mapsto (\texttt{bundle}\ (v_1, \overrightarrow{v_1})), \ldots, \overrightarrow{x_n} \mapsto (\texttt{bundle}\ (v_n, \overrightarrow{v_n}))\}, \overrightarrow{(\lambda x\ e)}\rangle$$

$$\texttt{bundle}\ ((v \rhd \overrightarrow{v'}), \overrightarrow{(v \rhd \overrightarrow{v'})}) \stackrel{\triangle}{=} ((\texttt{bundle}\ (v, \overrightarrow{v'})) \rhd (\texttt{bundle}\ (\overrightarrow{v'}, \overrightarrow{\overrightarrow{v'}})))$$

$$\texttt{bundle}\ ((v_1, v_2), \overrightarrow{(v_1, v_2)}) \stackrel{\triangle}{=} ((\texttt{bundle}\ (v_1, \overrightarrow{v_1})), (\texttt{bundle}\ (v_2, \overrightarrow{v_2})))$$

The primitive `primal` is defined as follows:

$$\texttt{primal}\ \overrightarrow{v} \stackrel{\triangle}{=} v \quad \text{when } v \text{ is a primitive}$$

$$\texttt{primal}\ \overrightarrow{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, (\lambda x\ e)\rangle} \stackrel{\triangle}{=} \langle \{x_1 \mapsto (\texttt{primal}\ \overrightarrow{v_1}), \ldots, x_n \mapsto (\texttt{primal}\ \overrightarrow{v_n})\}, (\lambda x\ e)\rangle$$

$$\texttt{primal}\ (v \rhd \overrightarrow{v'}) \stackrel{\triangle}{=} v$$

$$\texttt{primal}\ \overrightarrow{(v_1, v_2)} \stackrel{\triangle}{=} ((\texttt{primal}\ \overrightarrow{v_1}), (\texttt{primal}\ \overrightarrow{v_2}))$$

The primitive `tangent` is defined as follows:

$$\texttt{tangent}\ \overrightarrow{v} \stackrel{\triangle}{=} \overrightarrow{v'} \quad \text{when } v \text{ is a primitive}$$

$$\texttt{tangent}\ \overrightarrow{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_2\}, (\lambda x\ e)\rangle} \stackrel{\triangle}{=} \langle \{x_1 \mapsto (\texttt{tangent}\ \overrightarrow{v_1}), \ldots, x_n \mapsto (\texttt{tangent}\ \overrightarrow{v_n})\}, (\lambda x\ e)\rangle$$

$$\texttt{tangent}\ (v \rhd \overrightarrow{v'}) \stackrel{\triangle}{=} \overrightarrow{v'}$$

$$\texttt{tangent}\ \overrightarrow{(v_1, v_2)} \stackrel{\triangle}{=} ((\texttt{tangent}\ \overrightarrow{v_1}), (\texttt{tangent}\ \overrightarrow{v_2}))$$

And the primitive `zero` is defined as follows:

$$\texttt{zero}\ v \stackrel{\triangle}{=} \overrightarrow{v'} \quad \text{when } v \text{ is a discrete scalar}$$

$$\texttt{zero}\ v \stackrel{\triangle}{=} \overrightarrow{0} \quad \text{when } v \in \mathbb{R}$$

$$\texttt{zero}\ \langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, (\lambda x\ e)\rangle \stackrel{\triangle}{=} \langle \{x_1 \mapsto (\texttt{zero}\ v_1), \ldots, x_n \mapsto (\texttt{zero}\ v_n)\}, (\lambda x\ e)\rangle$$

$$\texttt{zero}\ (v \rhd \overrightarrow{v'}) \stackrel{\triangle}{=} ((\texttt{zero}\ v) \rhd (\texttt{zero}\ \overrightarrow{v'}))$$

$$\texttt{zero}\ (v_1, v_2) \stackrel{\triangle}{=} ((\texttt{zero}\ v_1), (\texttt{zero}\ v_2))$$

Note the reflection on closure environments that occurs in all four of the above primitives. Also note the reflective transformation that is performed on the closure expressions. While the former falls within the conceptual framework of `map-closure`, the latter transcends that framework.

5. Flow Analysis

STALIN∇ performs a polyvariant union-free flow analysis using a formulation based on abstract interpretation. For expository purposes, in the following overview, we omit many details and, at times, give a simplified presentation that differs in technicalities, but not in spirit, from the actual implementation. *Inter alia*, we omit discussion of `letrec`, bundled pairs, and primitives.

5.1 Concrete Values and Environments

A *concrete value* v is either a concrete scalar or a concrete aggregate. A *concrete environment* σ is a (partial) map from variables to concrete values, represented extensionally as a set of bindings $x \mapsto v$. Let $\mathbb{B}$ denote $\{\texttt{\#t}, \texttt{\#f}\}$. A *concrete scalar* is either `()`, a *concrete boolean* $b \in \mathbb{B}$, a *concrete real* $r \in \mathbb{R}$, or a *concrete primitive* p. A *concrete aggregate* is either a *concrete closure* $\langle \sigma, e \rangle$, a *concrete bundle* $(v_1 \rhd v_2)$, or a *concrete pair* (v_1, v_2). We assume that the concrete environment of a concrete closure maps precisely the free variables of the expression of that concrete closure. A *concrete function* is either a concrete primitive or a concrete closure. We use $\top$ to refer to the set of all concrete values. We often omit the specifier 'concrete' when it is clear from context.

5.2 Concrete Equivalence

Our formulation of flow analysis requires notions of equivalence for expressions, concrete values, and concrete environments. Languages typically do not define equivalence for function values. We need such a notion of equivalence for flow analysis since abstract values and environments denote sets of concrete values and environments and flow analysis is formulated in terms of unions of such sets, and equality relations between such sets, which in turn requires a notion of equivalence between the members of such sets.

Flow analysis typically formulates expression equivalence as equivalence between indices assigned to source-code expressions. This is suitable only in the traditional case where the source program is fixed and explicitly available, in its entirety, prior to the start of flow analysis. Here, application of the AD primitives `bundle`, `primal`, and `tangent` creates new expressions via the transformation $\overrightarrow{e}$ (and its inverse), at least conceptually. Thus we instead

use a structural notion of expression equivalence, because in VLAD some expressions are not explicitly available prior to the start of flow analysis and are created during the process of flow analysis.

Expression, value, and environment equivalence are mutual notions. Nominally, expression, environment, and function equivalence are extensional: two expressions are equivalent if they evaluate to equivalent values in equivalent environments, two environments are equivalent if they map equivalent variables to equivalent values, and two functions are equivalent if they yield equivalent result values when applied to equivalent argument values. Equivalence for other values is structural. The extensional notion of expression, environment, and function equivalence is undecidable. Thus we adopt the following conservative approximation. We take two expressions to be equivalent if they are structurally equivalent, take two environments to be equivalent if they map equivalent variables to equivalent values, take primitives to be equivalent only to themselves, and take two closures to be equivalent if they contain equivalent expressions and environments. While we do not currently do so, one can strengthen this approximation with a suitable notion of α-equivalence.

5.3 Concrete Evaluation

We develop our abstract evaluator by modifying the following standard eval/apply *concrete evaluator*:

$$\begin{array}{lcl}
\mathcal{A}\ \langle\sigma, (\lambda x\ e)\rangle\ v_2 & \triangleq & \mathcal{E}\ e\ \sigma[x \mapsto v_2] \\
\mathcal{E}\ x\ \sigma & \triangleq & \sigma\ x \\
\mathcal{E}\ (\lambda x\ e)\ \sigma & \triangleq & \langle\sigma, (\lambda x\ e)\rangle \\
\mathcal{E}\ (e_1\ e_2)\ \sigma & \triangleq & \mathcal{A}\ (\mathcal{E}\ e_1\ \sigma)\ (\mathcal{E}\ e_2\ \sigma) \\
\mathcal{E}\ (e_1, e_2)\ \sigma & \triangleq & ((\mathcal{E}\ e_1\ \sigma), (\mathcal{E}\ e_2\ \sigma))
\end{array}$$

The above, however, does not enforce the constraint that the concrete environment of a concrete closure map precisely the free variables of the expression of that concrete closure. We can enforce this constraint, as well as the constraint that σ map precisely the free variables in e in any call $(\mathcal{E}\ e\ \sigma)$, by judiciously restricting the domains of concrete environments at various places in the above evaluator. So as not to obscure the presentation of our formulation, we omit such restriction operations both above and in similar situations for the remainder of the paper.

A *concrete analysis* a is a finite extensional partial representation of the concrete evaluator as a set of bindings $e \mapsto \sigma \mapsto v$. A concrete analysis a is *sound* if for every $(e \mapsto \sigma \mapsto v) \in a$, $v = (\mathcal{E}\ e\ \sigma)$.

5.4 Abstract Values and Environments

Most standard approaches to flow analysis take the space of abstract values to include unions. This is because they are typically applied to languages whose execution model supports tags and tag dispatching. Since we wish to compile code to a FORTRAN-like execution model that does not support tags and tag dispatching, our space of abstract values does not include unions.

Preclusion of unions further precludes recursive abstract values as such recursion could not terminate. As a consequence, all of our abstract values will correspond to data structures of fixed size and shape in the execution model. This allows our code generator to unbox all aggregate data.

An *abstract value* $\overline{v}$ is either an abstract scalar or an abstract aggregate. An *abstract environment* $\overline{\sigma}$ is a (partial) map from variables to abstract values, represented extensionally as a set of bindings $x \mapsto \overline{v}$. An *abstract scalar* is either a concrete scalar, an abstract boolean $\overline{\mathbb{B}}$, or an abstract real $\overline{\mathbb{R}}$. An *abstract aggregate* is either an *abstract closure* $\langle\overline{\sigma}, e\rangle$, an *abstract bundle* $(\overline{v}_1 \rhd \overline{v}_2)$, an *abstract pair* $(\overline{v}_1, \overline{v}_2)$, or an *abstract top* $\overline{\top}$. We assume that the abstract environment of an abstract closure maps precisely the free variables of the expression of that abstract closure. An *abstract function* is either a concrete primitive or an abstract closure.

Abstract values and environments denote their *extensions*, sets of concrete values and environments:

$$\begin{array}{ll}
\textsc{Ext}\ v & = \{v\} \\
\textsc{Ext}\ \overline{\mathbb{B}} & = \mathbb{B} \\
\textsc{Ext}\ \overline{\mathbb{R}} & = \mathbb{R} \\
\textsc{Ext}\ \langle\overline{\sigma}, e\rangle & = \bigcup_{\sigma \in (\textsc{Ext}\ \overline{\sigma})} \{\langle\sigma, e\rangle\} \\
\textsc{Ext}\ (\overline{v}_1 \rhd \overline{v}_2) & = \bigcup_{v_1 \in (\textsc{Ext}\ \overline{v}_1)} \bigcup_{v_2 \in (\textsc{Ext}\ \overline{v}_2)} \{(v_1 \rhd v_2)\} \\
\textsc{Ext}\ (\overline{v}_1, \overline{v}_2) & = \bigcup_{v_1 \in (\textsc{Ext}\ \overline{v}_1)} \bigcup_{v_2 \in (\textsc{Ext}\ \overline{v}_2)} \{(v_1, v_2)\} \\
\textsc{Ext}\ \overline{\top} & = \top \\
\multicolumn{2}{l}{\textsc{Ext}\ \{x_1 \mapsto \overline{v}_1, \ldots, x_n \mapsto \overline{v}_n\}} \\
\multicolumn{2}{l}{\quad = \bigcup_{v_1 \in (\textsc{Ext}\ \overline{v}_1)} \cdots \bigcup_{v_n \in (\textsc{Ext}\ \overline{v}_n)} \{\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\}}
\end{array}$$

When $(\textsc{Ext}\ \overline{v}) \subset (\textsc{Ext}\ \overline{v}')$ we say that $\overline{v}'$ is *wider* that $\overline{v}$.

5.5 Abstract Equivalence and Union

Our formulation of flow analysis uses notions of equivalence between abstract values and environments as well as unions of abstract values and environments. We take the equivalence relation between two abstract values or two abstract environments to denote the corresponding relation between their extensions. These relations can be determined precisely:

- $v = v$
- $\langle\overline{\sigma}, e\rangle = \langle\overline{\sigma}', e\rangle$ when $\overline{\sigma} = \overline{\sigma}'$
- $(\overline{v}_1 \rhd \overline{v}_2) = (\overline{v}_1' \rhd \overline{v}_2')$ when $(\overline{v}_1 = \overline{v}_1') \wedge (\overline{v}_2 = \overline{v}_2')$
- $(\overline{v}_1, \overline{v}_2) = (\overline{v}_1', \overline{v}_2')$ when $(\overline{v}_1 = \overline{v}_1') \wedge (\overline{v}_2 = \overline{v}_2')$
- $\{x_1 \mapsto \overline{v}_1, \ldots, x_n \mapsto \overline{v}_n\} = \{x_1 \mapsto \overline{v}_1', \ldots, x_n \mapsto \overline{v}_n'\}$ when $(\overline{v}_1 = \overline{v}_1') \wedge \cdots \wedge (\overline{v}_n = \overline{v}_n')$

We take the union of two abstract values or two abstract environments to denote the abstract value or the abstract environment whose extension is the union of the extensions of those two abstract values or two abstract environments. Such an abstract value or abstract environment may not exist. We compute a conservative approximation to this notion, widening the result if necessary:

- $v \cup v \Longrightarrow v$
- $b_1 \cup b_2 \Longrightarrow \overline{\mathbb{B}}$ when $b_1 \neq b_2$
- $b \cup \overline{\mathbb{B}} \Longrightarrow \overline{\mathbb{B}}$
- $r_1 \cup r_2 \Longrightarrow \overline{\mathbb{R}}$ when $r_1 \neq r_2$
- $r \cup \overline{\mathbb{R}} \Longrightarrow \overline{\mathbb{R}}$
- $\langle\overline{\sigma}, e\rangle \cup \langle\overline{\sigma}', e\rangle \Longrightarrow \langle(\overline{\sigma} \cup \overline{\sigma}'), e\rangle$
- $(\overline{v}_1 \rhd \overline{v}_2) \cup (\overline{v}_1' \rhd \overline{v}_2') \Longrightarrow ((\overline{v}_1 \cup \overline{v}_1') \rhd (\overline{v}_2 \cup \overline{v}_2'))$
- $(\overline{v}_1, \overline{v}_2) \cup (\overline{v}_1', \overline{v}_2') \Longrightarrow ((\overline{v}_1 \cup \overline{v}_1'), (\overline{v}_2 \cup \overline{v}_2'))$
- $\{x_1 \mapsto \overline{v}_1, \ldots, x_n \mapsto \overline{v}_n\} \cup \{x_1 \mapsto \overline{v}_1', \ldots, x_n \mapsto \overline{v}_n'\}$ $\Longrightarrow \{x_1 \mapsto (\overline{v}_1 \cup \overline{v}_1'), \ldots, x_n \mapsto (\overline{v}_n \cup \overline{v}_n')\}$
- otherwise return $\overline{\top}$

5.6 Abstract Evaluation

An *abstract analysis* $\overline{a}$ is a set of bindings $e \mapsto \overline{\sigma} \mapsto \overline{v}$. The *extension* of an abstract analysis $\overline{a}$ is the following set of concrete analyses:

$$\mathbf{cf}\ \left\{ \left\{ e \mapsto \sigma \mapsto v \;\middle|\; \left((\exists \overline{\sigma}, \overline{v})\ \begin{pmatrix} ((e \mapsto \overline{\sigma} \mapsto \overline{v}) \in \overline{a}) \wedge \\ (\sigma \in (\textsc{Ext}\ \overline{\sigma})) \wedge \\ (v \in (\textsc{Ext}\ \overline{v})) \end{pmatrix} \right) \right\} \;\middle|\; \left((\exists \overline{\sigma}, \overline{v})\ \begin{pmatrix} ((e \mapsto \overline{\sigma} \mapsto \overline{v}) \in \overline{a}) \wedge \\ (\sigma \in (\textsc{Ext}\ \overline{\sigma})) \end{pmatrix} \right) \right\}$$

where **cf** denotes the set-theoretic choice function, the function that maps a set s_1 of sets to a set s_2 of all sets that contain one member from each member of s_1. An abstract analysis is *sound* if it contains a sound concrete analysis in its extension.

We need a notion of equivalence for abstract analyses to define the fixpoint of abstract interpretation. Nominally, two abstract analyses are equivalent if their extensions are equivalent. We conservatively approximate this by taking two bindings to be equivalent if their corresponding expressions, abstract environments, and abstract values are equivalent and take two abstract analyses to be equivalent if they contain equivalent bindings. We compute an abstract analysis with the following *abstract evaluator*:

$$\overline{\mathcal{E}}_1\ e\ \overline{\sigma}\ \overline{a} \triangleq \begin{cases} \overline{v} & \text{when } (e \mapsto \overline{\sigma} \mapsto \overline{v}) \in \overline{a} \\ \overline{\top} & \text{otherwise} \end{cases}$$

$$\overline{\mathcal{A}}\ \langle \overline{\sigma}, (\lambda x\ e)\rangle\ \overline{v}_2\ \overline{a} \triangleq \begin{cases} \overline{\mathcal{E}}_1\ e\ \overline{\sigma}[x \mapsto \overline{v}_2]\ \overline{a} & \text{when } \overline{v}_2 \neq \overline{\top} \\ \overline{\top} & \text{otherwise} \end{cases}$$

$$\overline{\mathcal{A}}\ \overline{\top}\ \overline{v}_2\ \overline{a} \triangleq \overline{\top}$$

$$\overline{\mathcal{E}}\ x\ \overline{\sigma}\ \overline{a} \triangleq \overline{\sigma}\ x$$

$$\overline{\mathcal{E}}\ (\lambda x\ e)\ \overline{\sigma}\ \overline{a} \triangleq \langle \overline{\sigma}, (\lambda x\ e)\rangle$$

$$\overline{\mathcal{E}}\ (e_1\ e_2)\ \overline{\sigma}\ \overline{a} \triangleq \overline{\mathcal{A}}\ (\overline{\mathcal{E}}_1\ e_1\ \overline{\sigma}\ \overline{a})\ (\overline{\mathcal{E}}_1\ e_2\ \overline{\sigma}\ \overline{a})\ \overline{a}$$

$$\overline{\mathcal{E}}\ (e_1, e_2)\ \overline{\sigma}\ \overline{a} \triangleq \begin{cases} ((\overline{\mathcal{E}}_1\ e_1\ \overline{\sigma}\ \overline{a}), (\overline{\mathcal{E}}_1\ e_2\ \overline{\sigma}\ \overline{a})) \\ \quad \text{when } ((\overline{\mathcal{E}}_1\ e_1\ \overline{\sigma}\ \overline{a}) \neq \overline{\top}) \\ \qquad \wedge\ ((\overline{\mathcal{E}}_1\ e_2\ \overline{\sigma}\ \overline{a}) \neq \overline{\top}) \\ \overline{\top} \qquad \text{otherwise} \end{cases}$$

$$\overline{\mathcal{E}}'_1\ e\ \overline{\sigma}\ \overline{a} \triangleq \begin{cases} \{e \mapsto \overline{\sigma} \mapsto \overline{\top}\} & \text{when } \neg(\exists \overline{v})(e \mapsto \overline{\sigma} \mapsto \overline{v}) \in \overline{a} \\ \{\} & \text{otherwise} \end{cases}$$

$$\overline{\mathcal{A}}'\ \langle \overline{\sigma}, (\lambda x\ e)\rangle\ \overline{v}_2\ \overline{a} \triangleq \begin{cases} \overline{\mathcal{E}}'_1\ e\ \overline{\sigma}[x \mapsto \overline{v}_2]\ \overline{a} & \text{when } \overline{v}_2 \neq \overline{\top} \\ \{\} & \text{otherwise} \end{cases}$$

$$\overline{\mathcal{A}}'\ \overline{\top}\ \overline{v}_2\ \overline{a} \triangleq \{\}$$

$$\overline{\mathcal{E}}'\ x\ \overline{\sigma}\ \overline{a} \triangleq \{\}$$

$$\overline{\mathcal{E}}'\ (\lambda x\ e)\ \overline{\sigma}\ \overline{a} \triangleq \{\}$$

$$\overline{\mathcal{E}}'\ (e_1\ e_2)\ \overline{\sigma}\ \overline{a} \triangleq (\overline{\mathcal{E}}'_1\ e_1\ \overline{\sigma}\ \overline{a}) \cup (\overline{\mathcal{E}}'_1\ e_2\ \overline{\sigma}\ \overline{a}) \cup (\overline{\mathcal{A}}'\ (\overline{\mathcal{E}}_1\ e_1\ \overline{\sigma}\ \overline{a})\ (\overline{\mathcal{E}}_1\ e_2\ \overline{\sigma}\ \overline{a})\ \overline{a})$$

$$\overline{\mathcal{E}}'\ (e_1, e_2)\ \overline{\sigma}\ \overline{a} \triangleq (\overline{\mathcal{E}}'_1\ e_1\ \overline{\sigma}\ \overline{a}) \cup (\overline{\mathcal{E}}'_1\ e_2\ \overline{\sigma}\ \overline{a})$$

$$\mathcal{U}\ \overline{a} \triangleq \bigcup_{(e \mapsto \overline{\sigma} \mapsto \overline{v}) \in \overline{a}} \{e \mapsto \overline{\sigma} \mapsto (\overline{\mathcal{E}}\ e\ \overline{\sigma}\ \overline{a})\} \cup (\overline{\mathcal{E}}'\ e\ \overline{\sigma}\ \overline{a})$$

We then compute $\overline{a}^* = \mathcal{U}^*\ \overline{a}_0$, where $\overline{a}_0 = \{e_0 \mapsto \sigma_0 \mapsto \overline{\top}\}$ is the initial abstract analysis, e_0 is the program, σ_0 is the basis, containing *inter alia* any bindings produced by constant conversion, and $\mathcal{U}^*$ is the least fixpoint of $\mathcal{U}$. The above flow-analysis procedure might not terminate, i.e., the least fixpoint might not exist. It is easy to see that the initial abstract analysis is sound and that $\mathcal{U}$ preserves soundness. Thus by induction, $\overline{a}^*$ is sound when it exists. The algorithm has the property that $\overline{\top}$ will never appear as the target of an abstract-environment binding or as a slot of an abstract aggregate value. The only place in an abstract analysis that $\overline{\top}$ can appear is as the target of a abstract-analysis binding, e.g., $e \mapsto \overline{\sigma} \mapsto \overline{\top}$. Our code generator only handles abstract analyses where $(\overline{\mathcal{E}}_1\ e\ \sigma\ \overline{a}^*) \neq \overline{\top}$ for all e and σ that would occur as arguments to $\mathcal{E}$ during a concrete evaluation $(\mathcal{E}\ e_0\ \sigma_0)$. We abort the compilation if this condition is violated. This can only occur when the union of two abstract values yields $\overline{\top}$. The only place where the union of two abstract values is computed is between the results of the consequent and alternate of `if-procedure`.

5.7 Imprecision Introduction

The above flow-analysis procedure yields a concrete analysis for any program e_0 that terminates. This is equivalent to running the program during flow analysis. To produce a non-concrete analysis, we add a primitive `real` to the basis that behaves like the identity function on reals during execution but yields $\overline{\mathbb{R}}$ during flow analysis. In the examples in Section 7, we judiciously annotate our code with a small number of calls to `real` around constants, so that the programs perform all of the same floating-point computation as the variants in other languages, but leave certain constants as concrete values so that flow analysis terminates and satisfies the non-$\overline{\top}$ condition discussed above.

6. Code Generation

STALIN∇ generates FORTRAN-like C code using an abstract analysis produced by polyvariant union-free flow analysis. In such an analysis, every application targets either a known primitive or a known lambda expression, potentially one created by flow-analysis-time source-code transformation induced by application of AD primitives. Recent versions of GCC will compile this C code to machine code similar to that generated by good FORTRAN compilers, given aggressive inlining, mediated by 'always inline' directives produced by our code generator, and scalar replacement of aggregates, enabled with the command-line option `--param sra-field-structure-ratio=0`. For expository purposes, in the following overview, we omit many details and, at times, give a simplified presentation that differs in technicalities, but not in spirit, from the actual implementation. *Inter alia*, we omit discussion of `letrec`, bundled pairs, and primitives.

Our code generator produces C code that is structurally isomorphic to the VLAD code. There is a C function for each specialized VLAD function, both closures and primitives. There is a function call in the C code for each application in each specialized closure expression. There are calls to constructor functions in the C code for each lambda expression and `cons` expression in each specialized closure expression. And there is C code that corresponds to each variable access in each specialized closure expression. The aggregate data is isomorphic as well. There is a C `struct` for each specialized aggregate datatype in the VLAD code, including closures, and a slot in that C `struct` for each corresponding slot in the VLAD object. (We adopt a flat closure representation. In the absence of mutation and `eq?`, as is the case for VLAD, all closure representations are extensionally equivalent and reduce to flat closures by unboxing.) One deviation from the above is that void `structs`, `struct` slots, arguments, and expressions are eliminated, as well as functions that return void results. The efficiency of the code generated results from polyvariant specialization, the union-free analysis, unboxing of all aggregate data, and aggressive inlining.

We assume a map $\mathcal{X}$ from alpha-converted VLAD variables to unique C identifiers, a map $\mathcal{S}$ from abstract values to unique C identifiers, and a map $\mathcal{F}$ from pairs of abstract values to unique C identifiers.

An abstract value is *void* when it does not contain any (nested) $\overline{\mathbb{B}}$ or $\overline{\mathbb{R}}$ values. Our code generator adopts the following map from

non-void abstract values to C specifiers:

$$\mathcal{T}\ \overline{v} \triangleq \begin{cases} \texttt{int} & \text{when } \overline{v} = \overline{\mathbb{B}} \\ \texttt{double} & \text{when } \overline{v} = \overline{\mathbb{R}} \\ \texttt{struct } (\mathcal{S}\ \overline{v}) & \\ \quad \text{where } \texttt{struct s}i\ \texttt{\{}(\mathcal{T}\ \overline{v}_1)\ (\mathcal{X}\ x_1)\texttt{;} & \\ \qquad \ldots\texttt{;} & \\ \qquad (\mathcal{T}\ \overline{v}_n)\ (\mathcal{X}\ x_n)\texttt{;\}}\texttt{;} & \\ \quad \text{when } \overline{v} = \langle \{x_1 \mapsto \overline{v}_1, \ldots, x_n \mapsto \overline{v}_n\}, e \rangle & \\ \quad \text{where } \texttt{struct } (\mathcal{S}\ \overline{v})\ \texttt{\{}(\mathcal{T}\ \overline{v}_1)\ \texttt{p;}\ (\mathcal{T}\ \overline{v}_2)\ \texttt{t;\}}\texttt{;} & \\ \quad \text{when } \overline{v} = (\overline{v}_1 \rhd \overline{v}_2) & \\ \quad \text{where } \texttt{struct } (\mathcal{S}\ \overline{v})\ \texttt{\{}(\mathcal{T}\ \overline{v}_1)\ \texttt{a;}\ (\mathcal{T}\ \overline{v}_2)\ \texttt{d;\}}\texttt{;} & \\ \quad \text{when } \overline{v} = (\overline{v}_1, \overline{v}_2) & \end{cases}$$

eliminating void `struct` slots. We also generate C constructor functions $(\mathcal{M}\ \overline{v})$ of the appropriate arity for each non-void abstract aggregate value $\overline{v}$.

Our code generator adopts the following map from VLAD expressions e that evaluate to non-void abstract values in the abstract environment $\overline{\sigma}$ to C expressions:

$$\mathcal{C}\ x\ \overline{\sigma} \triangleq \begin{cases} (\mathcal{X}\ x) & \text{when } x \text{ is bound} \\ \texttt{c.}(\mathcal{X}\ x) & \text{when } x \text{ is free} \end{cases}$$

$$\mathcal{C}\ (\lambda x\ e)\ \overline{\sigma} \triangleq \text{a call to } (\mathcal{M}\ (\overline{\mathcal{E}}_1\ (\lambda x\ e)\ \overline{\sigma}\ \overline{a}^*)) \text{ with arguments that have the form of variable accesses}$$

$$\mathcal{C}\ (e_1\ e_2)\ \overline{\sigma} \triangleq (\mathcal{F}\ (\overline{\mathcal{E}}_1\ e_1\ \overline{\sigma}\ \overline{a}^*)\ (\overline{\mathcal{E}}_1\ e_2\ \overline{\sigma}\ \overline{a}^*))((\mathcal{C}\ e_1\ \overline{\sigma}),\ (\mathcal{C}\ e_2\ \overline{\sigma}))$$

$$\mathcal{C}\ (e_1, e_2)\ \overline{\sigma} \triangleq (\mathcal{M}\ ((\overline{\mathcal{E}}_1\ e_1\ \overline{\sigma}\ \overline{a}^*), (\overline{\mathcal{E}}_1\ e_2\ \overline{\sigma}\ \overline{a}^*)))((\mathcal{C}\ e_1\ \overline{\sigma}),\ (\mathcal{C}\ e_2\ \overline{\sigma}))$$

eliminating void arguments.

Our code generator generates distinct C functions for each abstract closure $\langle \overline{\sigma}, (\lambda x\ \ e) \rangle$ that yields a non-void abstract value when called on each abstract value $\overline{v}$:

$$(\mathcal{T}\ (\overline{\mathcal{A}}\ \langle \overline{\sigma}, (\lambda x\ e) \rangle\ \overline{v}\ \overline{a}^*))$$
$$(\mathcal{F}\ \langle \overline{\sigma}, (\lambda x\ e) \rangle\ \overline{v})\texttt{(}(\mathcal{T}\ \langle \overline{\sigma}, (\lambda x\ e) \rangle)\ \texttt{c,}\ (\mathcal{T}\ \overline{v})\ (\mathcal{X}\ x))$$
$$\texttt{\{return}\ (\mathcal{C}\ \overline{\sigma}[x \mapsto \overline{v}]\ e)\texttt{;\}}$$

eliminating void parameters. Finally, we generate the entry point:

$$\texttt{int main(void)\{}(\mathcal{C}\ e_0\ \overline{\sigma}_0)\texttt{;return 0;\}}$$

For expository purposes, we omit discussion of the generation of C functions for primitives and constructors. We generate 'always inline' directives on all generated C functions, including those generated for primitives and constructors, except for `main` and those selected to break cycles in the call graph.

With a polyvariant union-free flow analysis, the target of every call site is known. This allows generating direct function calls or inlined primitives for each call site. Calls to the AD primitives involve nothing more than rearrangements of (aggregate) data structures from one known fixed shape to another known fixed shape. As aggregate data is unboxed and calls to primitives are inlined, this is usually compiled away.

7. Examples

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

Figure 1 gives the essence of the two examples. It starts with code shared between these examples: `multivariate-argmin` implements a multivariate optimizer using adaptive naïve gradient descent. This iterates $\mathbf{x}_{i+1} = \eta \nabla f\ \mathbf{x}_i$ until either $\|\nabla f\ \mathbf{x}\|$ or $\|\mathbf{x}_{i+1} - \mathbf{x}_i\|$ is small, increasing η when progress is made and decreasing η when no progress is made. Omitted are definitions for standard SCHEME primitives and the functions `sqr` that squares its argument, `map-n` that maps a function over the list `(0 ... n - 1)`, `reduce` that folds a binary function with a specified identity over a list, `v+` and `v-` that perform vector addition and subtraction, `k*v` that multiplies a vector by a scalar, `magnitude` that computes the magnitude of a vector, `distance` that computes the l^2 norm of two vectors, and `e` that returns the i-th basis vector of dimension n.

The first example, `saddle`, computes a saddle point:

$$\min_{(x_1,y_1)} \max_{(x_2,y_2)} ({x_1}^2 + {y_1}^2) - ({x_2}^2 + {y_2}^2)$$

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

Naïve Euler ODE integration:

$$\begin{aligned} \ddot{\mathbf{x}}(t) &= -\nabla_{\mathbf{x}}\, p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)} \\ \dot{\mathbf{x}}(t + \Delta t) &= \dot{\mathbf{x}}(t) + \Delta t\, \ddot{\mathbf{x}}(t) \\ \mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t) \end{aligned}$$

is used to compute the particle's path, with a linear interpolation to find the x-axis intersect:

$$\begin{aligned} \text{When } x_1(t + \Delta t) &\leq 0 \\ \text{let: } \Delta t_f &= -x_1(t)/\dot{x}_1(t) \\ t_f &= t + \Delta t_f \\ \mathbf{x}(t_f) &= \mathbf{x}(t) + \Delta t_f \dot{\mathbf{x}}(t) \\ \text{Error: } E(w) &= x_0(t_f)^2 \end{aligned}$$

E is minimized with respect to w by `multivariate-argmin`.

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

STALIN∇ performed a polyvariant union-free flow analysis on both of these examples, and generated FORTRAN-like code. Variants of these examples were also coded in SCHEME, ML, HASKELL, C++, and FORTRAN, and run with a variety of compilers and AD implementations. For SCHEME, we used two implementations of forward AD. When compiling with MIT SCHEME

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

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

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

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

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

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

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

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

we used SCMUTILS (Sussman et al. 2001) and when compiling with IKARUS, STALIN, SCHEME->C, CHICKEN, BIGLOO, and LARCENY we used a custom implementation of forward AD. For ML, we used a translation of the latter and compiled with MLTON and OCAML. For HASKELL, we used a simplified non-tower version of the forward AD method of Karczmarczuk (2001) and compiled with GHC. For C++, we used the FADBAD++ implementation of forward AD and compiled with G++. For FORTRAN, we used both the ADIFOR and TAPENADE implementations of forward AD and compiled with G77. In all of the variants, we attempted to be faithful to both the generality of the mathematical concepts represented in the examples and to the standard coding style typically used for each particular language. In other words, to make the examples fair, we coded the ML variants of the examples the way an ML programmer would, the HASKELL variants the way a HASKELL programmer would, etc.

Note that ADIFOR and TAPENADE are transformation systems based on a preprocessor. Except for STALIN∇, the remaining systems use an overloading approach. Also note that SCMUTILS, FADBAD++, ADIFOR, and TAPENADE are existing AD implementations developed by others. While the remainder were written by us, they use standard methods. Finally note that all of these variants were compiled and run as machine code. In the case of MZSCHEME, this was done by a JIT compiler; for all the others this was done by ahead-of-time compilation, potentially via C. Many of these compilers are well acknowledged as being among the most sophisticated and highly optimizing compilers in existence today.

Many of the variants of our examples require that the source program be written in an unconventional fashion. Furthermore, many require manual modification of the preprocessor output. Many SCHEME implementations do not allow redefinition of builtin arithmetic procedures. Thus the SCHEME variants of our examples are written using alternate arithmetic procedures `d+`, `d-`, etc. SCMUTILS incorrectly implements the overloaded implementation of `=`. Thus the SCMUTILS variants of our examples are written using an alternate procedure `d=`. It is impossible to implement forward AD in ML in a fashion that applies to unmodified source code. Thus the ML variants of our examples require that (a) all code which implements the function whose derivative is taken, including all code called by such code, be syntactically nested inside the redefinition of the primitives and (b) all real constants in that code be syntactically wrapped with the `BASE` constructor. It is conjectured to be impossible to implement nestable forward AD in HASKELL in a fashion that applies to unmodified source code. Thus the HASKELL variants of our examples require that the code that implements the functions whose derivatives are taken be manually annotated with appropriate calls to `lift` to properly handle nesting. Code must be written with templates in FADBAD++ to support taking derivatives of different orders. Code that is transformed by TAPENADE multiple times must be post-edited. The flow analysis used by ADIFOR yields incorrect derivative code that produces the wrong answer, without warning, when using the same file organization as all of the other variants, where each example is contained in a single file. Thus the ADIFOR variants of our examples circumvent this flaw by manually partitioning each example into three files that contain the code that is transformed zero, one, and two times. Finally, TAPENADE cannot transform code that relies on indirect (i.e., external) subroutine calls. Thus the TAPENADE variants of our examples require manual specialization of the `multivariate_argmin` subroutine. These AD-specific limitations of existing languages and systems are in addition to the standard limitations of languages like C++ and FORTRAN relative to higher-order functional-programming languages. For example, since our examples make use of nested lambda expressions with lexically-scoped free variables to implement the nested minimax optimization in `saddle` and the potential function in `particle`, the FADBAD++, ADIFOR, and TAPENADE variants of our examples manually implement the requisite closures by way of global variables in C++ and common blocks in FORTRAN.

Table 1 summarizes the run times of our examples normalized relative to a unit run time for STALIN∇.[4] Note that STALIN∇ exhibits an increase in performance of one to three orders of magnitude when compared with the overloading-based forward AD implementations for both functional and imperative languages and matches the performance of the transformation-based forward AD implementations for imperative languages.

The different variants do not perform the exact same floating point computation graph. First, FADBAD++, ADIFOR, and TAPENADE support tangent-vector mode which can compute multiple tangent values with a single primal value, which allows them to compute gradients with fewer redundant primal calculations. Second, the implementations of `multivariate_argmin` in FORTRAN and C++ return both the location of the local optimum and the value of the objective function at that location, while the others use a redundant extra evaluation of the objective function to determine its value at the local optimum. Finally, the implementation of `multivariate_argmin` in C++ determines the value of the objective function as a byproduct of computing the gradient of the objective function, while the other implementations perform redundant computation to determine that value. These factors bias the performance measurements in favor of FADBAD++, ADIFOR, and TAPENADE and can account for the performance difference of ADIFOR and TAPENADE *vs.* STALIN∇.

It is difficult to conduct meaningful performance comparisons of multiple implementations of forward AD across multiple implementations of multiple programming languages. Changes in coding style, compiler options, etc., are likely to affect the run times by perhaps a few tens of percent. Thus we should not draw conclusions from the particular measured run times. However, it seems unlikely that changes in coding style, compiler options, and the like would improve run times by an order of magnitude. It therefore appears significant that STALIN∇ exhibits approximately the same performance as ADIFOR and TAPENADE and outperforms all of the other systems, on both examples, by approximately one to three orders of magnitude.

8. Discussion

Early transformation-based AD implementations were simple, but produced inefficient transformed source code. As the field of AD evolved, transformation-based systems started employing increasingly sophisticated analyses and transformations with the goal of producing efficient transformed source code. These analyses and transformations mirror the analyses and transformations done by compilers, leading to tremendous redundancy. Furthermore, the AD community is slowly and systematically rediscovering techniques that have been known in the compiler community for years, reimplementing them in various combinations for AD transformers for different programming languages. This motivates including AD in the compiler; in fact there is a commercial effort to incorporate AD into the NAG FORTRAN compiler (Naumann and Riehme 2005).

Since at least as far back as Steele and Sussman (1976), the programming-language community has realized the benefits of using variants of the lambda calculus as compiler intermediate languages. However, the AD community has not yet adopted the lambda calculus as the intermediate language for transformation-based AD. In contrast, the approach we advocate is illustrated in Figure 2. VLAD is but one potential front end that is nothing more than syntactic sugar for the untyped lambda calculus. Front ends for other languages can be constructed given a suitable extension of the intermediate language. Both AD transformations and compiler analyses and optimizations apply to this intermediate language. Back ends can be constructed to generate either machine code or source code in a variety of languages. This allows the common core of AD techniques to be shared among AD implementations for multiple languages either as part of a compiler or as part of a traditional transformation-based preprocessor.

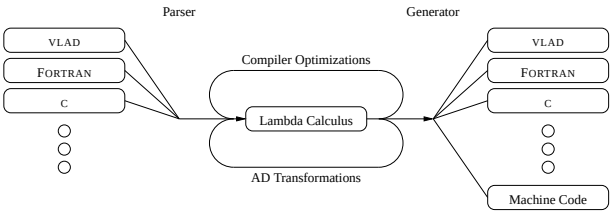

Figure 2. Our vision: using (a suitable extension of) the lambda calculus as a unified intermediate language that supports both compiler optimizations and AD transformations for a variety of source and target languages. This allows AD to incorporate known compiler optimizations and allows effort to build efficient AD to be shared among different languages.

We have embodied the ideas in this paper in a research prototype compiler sufficient to demonstrate their power and feasibility. Although it is not a production-quality compiler (it is slow, cannot handle large examples, does not support arrays or other update-in-place data structures, and is in general unsuitable for end users) remedying its deficiencies and building a production-quality compiler would be straightforward, involving only known methods (Nielson et al. 1999; Wadler 1990). The limitation to union-free analyses and finite unrolling of recursive data structures could also be relaxed using standard implementation techniques.

9. Novelty and Significance

This paper makes two novel contributions:

(1) A novel set of higher-order functions (`j*`, `primal`, `tangent`, `bundle`, and `zero`) for performing forward AD in a functional language using source-to-source transformation via run-time reflection. This is the focus of Sections 3 and 4.

(2) A novel approach for using polyvariant flow analysis to eliminate such run-time reflection along with all other non-numerical scaffolding. This is the focus of Sections 5 and 6.

We wish to make it completely clear that the contribution here is neither the forward AD transformation (Wengert 1964) nor the general idea of polyvariant flow analysis (Shivers 1988).

The above contributions are significant because they support forward 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.

Acknowledgments

This work was supported, in part, by NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and a grant from the Higher Education Authority of Ireland. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies. Shawn Brownfield wrote an early version of the flow-analysis code and aided in formulating some of the algorithms used therein.

[4] `http://www.bcl.hamilton.ie/~qobi/tr-08-01/` contains the source code for all variants of our examples, the scripts used to produce Table 1, and the log produced by running those scripts. **Reviewer warning: target page is not blinded.**

	Language/Implementation																	
Example	STALIN∇	IKARUS	STALIN	SCHEME->C	CHICKEN	BIGLOO	GAMBIT	LARCENY	MZC	MZSCHEME	SCMUTILS	MLTON	OCAML	GHC	FADBAD++	ADIFOR	TAPENADE
`saddle`	1.00	59.12	95.42	111.86	231.03	155.01	129.90	190.62	611.38	719.82	715.71	11.18	21.23	31.04	5.93	0.49	0.72
`particle`	1.00	148.92	244.48	311.56	608.33	427.49	351.58	564.34	1450.07	1869.64	1505.52	33.35	59.31	75.00	32.09	0.85	1.76

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

References

C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, Aug. 1996.

C. H. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, 1941.

W. K. Clifford. Preliminary sketch of bi-quaternions. *Proceedings of the London Mathematical Society*, 4:381–95, 1873.

W. Clinger and J. Rees. *Revised*^{4} *Report on the Algorithmic Language* SCHEME, Nov. 1991.

L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis, 2004. URL `http://www.inria.fr/rrrt/rt-0300.html`.

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

U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4), 2005.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, New York, 1999.

O. G. 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.

J. M. Siskind and B. A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *Proceedings of the 2007 Symposium on Principles of Programming Languages*, pages 71–6, Nice, France, Jan. 2007.

B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Jan. 1980.

G. L. Steele, Jr. and G. J. Sussman. Lambda, the ultimate imperative. A. I. Memo 353, MIT Artificial Intelligence Laboratory, Mar. 1976.

G. J. Sussman, J. Wisdom, and M. E. Mayer. *Structure and Interpretation of Classical Mechanics*. MIT Press, Cambridge, MA, 2001.

P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on* LISP *and Functional Programming*, pages 61–78, Nice, France, 1990.

R. E. Wengert. A simple automatic derivative evaluation program. *Comm. of the ACM*, 7(8):463–4, 1964.