Binomial Checkpointing for Arbitrary Programs with No User Annotation*

Jeffrey Mark Siskind† Barak A. Pearlmutter‡

April 2016

Heretofore, automatic checkpointing at procedure-call boundaries [1], to reduce the space complexity of reverse mode, has been provided by systems like TAPENADE [2]. However, binomial checkpointing, or treeverse [3], has only been provided in AD systems in special cases, *e.g.*, through user-provided pragmas on DO loops in TAPENADE, or as the nested taping mechanism in ADOL-C for time integration processes, which requires that user code be refactored. We present a framework for applying binomial checkpointing to arbitrary code with no special annotation or refactoring required. This is accomplished by applying binomial checkpointing directly to a program trace. This trace is produced by a general-purpose checkpointing mechanism that is orthogonal to AD.

Listing 1: FORTRAN example

```
      subroutine f(x, y)
      n = 100003
      y = x
c$ad binomial-ckp n+1 30 1
      do i = 1, n
         m = l(x, i)
         do j = 1, m
            y = y*y
            y = sqrt(y)
         end do
      end do
      end
```

Consider the code fragment in Listing 1. This example, $y = f(x)$, while contrived, is a simple caricature of a situation that arises commonly in practice, *e.g.*, in adaptive grid methods. Here, the duration of the inner loop varies wildly as some function $l(x,i)$ of the input and the outer loop index, perhaps $2^{\lfloor \lg(n)\rfloor - \lfloor \lg(1+(1007\lfloor 3^x\rfloor i \bmod n))\rfloor}$, that is small on most iterations of the outer loop but $O(n)$ on a few iterations. Thus the optimality of the binomial schedule is violated. The issue is that the optimality of the binomial schedule holds at the level of primitive atomic computations but this is not reflected in the static syntactic structure of the source code. Often, the user is unaware or even unconcerned with the micro-level structure of atomic computations and does not wish to break the modularity of the source code to expose such. Yet the user may still wish to reap the benefits of an optimal binomial checkpointing schedule [4]. Moreover, the relative duration of different paths through a program may vary from loop iteration to loop iteration in a fashion that is data dependent, as shown by the above example, and not even statically determinable. We present an implementation strategy for checkpointing that does not require user placement of checkpoints and does not constrain checkpoints to subroutine boundaries, DO loops, or other syntactic program constructs. Instead, it can automatically and dynamically introduce a checkpoint at an arbitrary point in the computation that need not correspond to a syntactic program unit.

We have previously introduced VLAD, a pure functional language with builtin AD operators for both forward and reverse mode. Here, we adopt slight variants of these operators with the following signatures.

$$\overrightarrow{\mathcal{J}} : f\ x\ \acute{x} \mapsto y\ \acute{y} \qquad \overleftarrow{\mathcal{J}} : f\ x\ \grave{y} \mapsto y\ \grave{x}$$

The $\overrightarrow{\mathcal{J}}$ operator calls a function f on a primal x with a tangent $\acute{x}$ to yield a primal y and a tangent $\acute{y}$. The $\overleftarrow{\mathcal{J}}$ operator calls a function f on a primal x with a cotangent $\grave{y}$ to yield a primal y and a cotangent $\grave{x}$. Here, we restrict ourselves to the case where (co)tangents are ground data values, *i.e.*, reals and (arbitrary) data structures containing reals and other scalar values, but not functions (*i.e.*, closures). For our purposes, the crucial aspect of the design is that the AD operators are provided within the language, since these provide the portal to the checkpointing mechanism.

$$\begin{aligned}
\mathcal{A}\ \langle(\lambda x.e),\rho\rangle\ v &= \mathcal{E}\ \rho[x\mapsto v]\ e\\
\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 &= \textbf{let}\ (v_4 \triangleright \acute{v}_5) = \mathcal{A}\ v_1\ (v_2 \triangleright \acute{v}_3)\ \textbf{in}\ (v_4,\acute{v}_5)\\
\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 &= \textbf{let}\ (v_4 \triangleleft \grave{v}_5) = ((\mathcal{A}\ v_1\ v_2) \triangleleft \grave{v}_3)\ \textbf{in}\ (v_4,\grave{v}_5)\\
\mathcal{E}\ \rho\ c &= c\\
\mathcal{E}\ \rho\ x &= \rho\ x\\
\mathcal{E}\ \rho\ (\lambda x.e) &= \langle(\lambda x.e),\rho\rangle\\
\mathcal{E}\ \rho\ (e_1\ e_2) &= \mathcal{A}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\\
\mathcal{E}\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) &= \textbf{if}\ (\mathcal{E}\ \rho\ e_1)\ \textbf{then}\ (\mathcal{E}\ \rho\ e_2)\ \textbf{else}\ (\mathcal{E}\ \rho\ e_3)\\
\mathcal{E}\ \rho\ (\diamond e) &= \diamond(\mathcal{E}\ \rho\ e)\\
\mathcal{E}\ \rho\ (e_1 \bullet e_2) &= (\mathcal{E}\ \rho\ e_1)\bullet(\mathcal{E}\ \rho\ e_2)\\
\mathcal{E}\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) &= \overrightarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)\\
\mathcal{E}\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) &= \overleftarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)
\end{aligned}$$

Figure 1: Direct-style evaluator for VLAD.

In previous work, we introduced STALIN∇, a highly optimizing compiler for VLAD. Here, we formulate a simple evaluator (interpreter) for VLAD (Fig. 1) and extend such to perform binomial checkpointing. The operators $\diamond$ and $\bullet$ range over the unary and binary basis functions respectively. This evaluator is written in what is known in the programming-language community as *direct style*, where functions (in this case $\mathcal{E}$, denoting 'eval', $\mathcal{A}$, denoting 'apply',

***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

and the implementations of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ in the host) take inputs as function-call arguments and yield outputs as function-call return values [5]. AD is performed by overloading the basis functions in the host, in a fashion similar to FADBAD++ [6], $x \triangleright \acute{x}$ denotes recursively bundling a data structure containing primals with a data structure containing tangents, or alternatively recursively unbundling such when used as a binder, and $y \triangleleft \grave{y}$ denotes running the reverse sweep on the tape y with the output cotangent $\grave{y}$, or alternatively extracting the primal y and input cotangent $\grave{x}$ from the tape when used as a binder $y \triangleleft \grave{x}$.

We introduce a new AD operator $\overset{\checkmark}{\mathcal{J}}$ to perform binomial checkpointing. The crucial aspect of the design is that the signature (and semantics) of $\overset{\checkmark}{\mathcal{J}}$ is *identical* to $\overleftarrow{\mathcal{J}}$; they are *completely interchangeable*, differing only in the space/time complexity tradeoffs. This means that code *need not be modified* to switch back and forth between ordinary reverse mode and binomial checkpointing, save interchanging calls to $\overleftarrow{\mathcal{J}}$ and $\overset{\checkmark}{\mathcal{J}}$.

Conceptually, the behavior of $\overset{\checkmark}{\mathcal{J}}$ is shown in Fig. 2. In this inductive definition, a function f is split into the composition of two functions g and h in step 1, the checkpoint u is computed by applying g to the input x in step 2, and the cotangent is computed by recursively applying $\overset{\checkmark}{\mathcal{J}}$ to h and g in steps 3 and 4. This divide-and-conquer behavior is terminated in a base case, when the function f is small, at which point the cotangent is computed with $\overleftarrow{\mathcal{J}}$, in step 0. If step 1 splits a function f into two functions g and h that take the same number of computational steps, the recursive divide-and-conquer process yields the logarithmic asymptotic space/time complexity of binomial checkpointing.

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case (f x fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(0)
inductive case:	$h \circ g = f$	(1)
	$u = g\ x$	(2)
	$(y, \grave{u}) = \overset{\checkmark}{\mathcal{J}}\ h\ u\ \grave{y}$	(3)
	$(u, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{u}$	(4)

Figure 2: Algorithm for binomial checkpointing.

The central difficulty in implementing the above is performing step 1, namely splitting a function f into two functions g and h, ideally ones that take the same number of computational steps. A sophisticated user can manually rewrite a subroutine f into two subroutines g and h. A sufficiently powerful compiler or source transformation tool might also be able to, with access to nonlocal program text. But an overloading system, with access only to local information, would not be able to.

We solve this problem by providing an interface to a general-purpose checkpointing mechanism orthogonal to AD.

PRIMOPS $f\ x \mapsto (y, n)$	Return $y = f(x)$ along with the number n of steps needed to compute y.
CHECKPOINT $f\ x\ n \mapsto u$	Run the first n steps of the computation of $f(x)$ and return a checkpoint u.
RESUME $u \mapsto y$	If $u =$ (CHECKPOINT $f\ x\ n$), return $y = f(x)$.

This interface allows (a) determining the number of steps of a computation, (b) interrupting a computation after a specified number of steps, usually half the number of steps determined by the mechanism in (a), and (c) resuming an interrupted computation to completion. A variety of implementation strategies for this interface are possible. We present one in detail momentarily and briefly discuss others below.

Irrespective of how one implements the general-purpose checkpointing interface, one can use it to implement $\overset{\checkmark}{\mathcal{J}}$ as shown in Fig. 3. The function f is split into the composition of two functions g and h by taking g as $\lambda x.$CHECKPOINT $f\ x\ n$, where n is half the number of steps determined by PRIMOPS $f\ x$, and h as $\lambda u.$RESUME u.

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case:	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(0)
inductive case:	$(y, 2n) =$ PRIMOPS $f\ x$	(1)
	$u =$ CHECKPOINT $f\ x\ n$	(2)
	$(y, \grave{u}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda u.\text{RESUME}\ u)\ u\ \grave{y}$	(3)
	$(u, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda x.\text{CHECKPOINT}\ f\ x\ n)\ x\ \grave{u}$	(4)

Figure 3: Binomial checkpointing via general checkpointing interface.

One way of implementing the general-purpose checkpointing interface is to convert the evaluator from direct style to continuation-passing style (CPS, [7]), where functions (in this case $\mathcal{E}$, $\mathcal{A}$, $\overrightarrow{\mathcal{J}}$, and $\overleftarrow{\mathcal{J}}$ in the host) take an additional continuation input k and instead of yielding outputs via function-call return, do so by calling the continuation with said output as arguments (Fig. 5). In such a style, functions never return; they just call their continuation. With tail-call merging, such corresponds to a computed `go to` and does not incur stack growth. This crucially allows the interruption process to actually return a checkpoint data structure containing the saved state of the evaluator, including its continuation, allowing the evaluation to be resumed by calling the evaluator with this saved state. This 'level shift' of return to calling a continuation allowing an actual return to constitute checkpointing interruption is analogous to the way backtracking is classically implemented in PROLOG, with success implemented as calling a continuation and failure implemented as actual return. In our case, we further instrument the evaluator to thread two values as inputs and outputs: the count n of the number of evaluation steps, which is incremented at each call to $\mathcal{E}$, and the limit l of the number of steps, after which a checkpointing interrupt is triggered.

With this CPS evaluator, it is possible to implement the general-purpose checkpointing interface (Fig. 4), not for programs in the host, but for programs in the target; hence our choice of formulating the implementation around an evaluator (interpreter).

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.(v, n)))\ 0\ \infty\ f\ x$$
$$\text{CHECKPOINT}\ f\ x\ n = \mathcal{A}\perp 0\ n\ f\ x$$
$$\text{RESUME}\ [\![k, l, \rho, e]\!] = \mathcal{E}\ k\ l\ \infty\ \rho\ e$$

Figure 4: Implementation of the general-purpose checkpointing interface using the CPS evaluator.

$$
\begin{aligned}
\mathcal{A}\;k\;n\;l\;\langle(\lambda x.e),\rho\rangle\;v &= \mathcal{E}\;k\;n\;l\;\rho[x\mapsto v]\;e\\
\overrightarrow{\mathcal{J}}\;k\;n\;l\;v_1\;v_2\;\acute{v}_3 &= \mathcal{A}\;(\lambda n\;l\;(v_4 \triangleright \acute{v}_5).k\;n\;l\;(v_4,\acute{v}_5))\;n\;l\;v_1\;(v_2 \triangleright \acute{v}_3)\\
\overleftarrow{\mathcal{J}}\;k\;n\;l\;v_1\;v_2\;\acute{v}_3 &= \mathcal{A}\;(\lambda n\;l\;v.\mathbf{let}\;(v_4 \triangleleft \acute{v}_5) = v \triangleleft \grave{v}_3\;\mathbf{in}\;k\;n\;l\;(v_4,\acute{v}_5))\;n\;l\;v_1\;v_2\\
\mathcal{E}\;k\;l\;l\;\rho\;e &= [\![k,l,\rho,e]\!]\\
\mathcal{E}\;k\;n\;l\;\rho\;c &= k\;(n+1)\;l\;c\\
\mathcal{E}\;k\;n\;l\;\rho\;x &= k\;(n+1)\;l\;(\rho\;x)\\
\mathcal{E}\;k\;n\;l\;\rho\;(\lambda x.e) &= k\;(n+1)\;l\;\langle(\lambda x.e),\rho\rangle\\
\mathcal{E}\;k\;n\;l\;\rho\;(e_1\;e_2) &= \mathcal{E}\;(\lambda n\;l\;v_1.(\mathcal{E}\;(\lambda n\;l\;v_2.(\mathcal{A}\;k\;n\;l\;v_1\;v_2))\;n\;l\;\rho\;e_2))\;(n+1)\;l\;\rho\;e_1\\
\mathcal{E}\;k\;n\;l\;\rho\;(\mathbf{if}\;e_1\;\mathbf{then}\;e_2\;\mathbf{else}\;e_3) &= \mathcal{E}\;(\lambda n\;l\;v_1.(\mathbf{if}\;v_1\;\mathbf{then}\;(\mathcal{E}\;k\;n\;l\;\rho\;e_2)\;\mathbf{else}\;(\mathcal{E}\;k\;n\;l\;\rho\;e_3)))\;(n+1)\;l\;\rho\;e_1\\
\mathcal{E}\;k\;n\;l\;\rho\;(\diamond e) &= \mathcal{E}\;(\lambda n\;l\;v.(k\;n\;l\;(\diamond v)))\;(n+1)\;l\;\rho\;e\\
\mathcal{E}\;k\;n\;l\;\rho\;(e_1 \bullet e_2) &= \mathcal{E}\;(\lambda n\;l\;v_1.(\mathcal{E}\;(\lambda n\;l\;v_2.(k\;n\;l\;(v_1 \bullet v_2)))\;n\;l\;\rho\;e_2))\;(n+1)\;l\;\rho\;e_1\\
\mathcal{E}\;k\;n\;l\;\rho\;(\overrightarrow{\mathcal{J}}\;e_1\;e_2\;e_3) &= \mathcal{E}\;(\lambda n\;l\;v_1.(\mathcal{E}\;(\lambda n\;l\;v_2.(\mathcal{E}\;(\lambda n\;l\;v_3.(\overrightarrow{\mathcal{J}}\;k\;n\;l\;v_1\;v_2\;v_3))\;n\;l\;\rho\;e_3))\;n\;l\;\rho\;e_2))(n+1)\;l\;\;\rho\;e_1\\
\mathcal{E}\;k\;n\;l\;\rho\;(\overleftarrow{\mathcal{J}}\;e_1\;e_2\;e_3) &= \mathcal{E}\;(\lambda n\;l\;v_1.(\mathcal{E}\;(\lambda n\;l\;v_2.(\mathcal{E}\;(\lambda n\;l\;v_3.(\overleftarrow{\mathcal{J}}\;k\;n\;l\;v_1\;v_2\;v_3))\;n\;l\;\rho\;e_3))\;n\;l\;\rho\;e_2))(n+1)\;l\;\rho\;e_1
\end{aligned}
$$

Figure 5: CPS evaluator for VLAD.

We remove this restriction below. The implementation of PRIMOPS calls the evaluator with no limit and simply counts the number of steps to completion. The implementation of CHECKPOINT calls the evaluator with a limit that must be smaller than that needed to complete so a checkpointing interrupt is forced and the checkpoint data structure $[\![k,l,\rho,e]\!]$ is returned. The implementation of RESUME calls the evaluator with arguments from the saved checkpoint data structure.

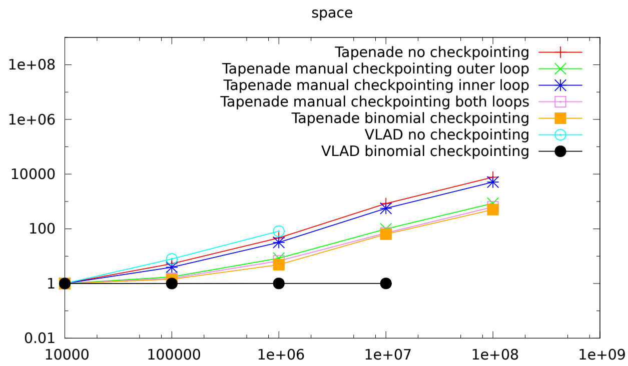
With this, it is possible to reformulate the Fortran example from Listing 1 in VLAD (Listing 2). Then one achieves binomial checkpointing simply by calling ($\overset{\checkmark}{\mathcal{J}}$ `f 3 1`).

Listing 2: VLAD example

```
(define (f x)
 (let ((n 100003))
  (let outer ((i 1) (y x))
   (if (> i n)
       y
       (outer (+ i 1)
              (let ((m (l x i)))
               (let inner ((j 1) (y y))
                (if (> j m)
                    y
                    (inner (+ j 1)
                           (sqrt (* y y))))))))))))
```

The efficacy of our method can be seen in the plots (Fig. 5) of the space and time usage, relative to that for the leftmost datapoint, of the above Fortran and VLAD examples with varying n. Tapenade was run without checkpointing, with manual checkpointing only around the body of the outer loop, with manual checkpointing only around the body of the inner loop, with manual checkpointing around the bodies of both loops, and with binomial checkpointing. VLAD was run with $\overleftarrow{\mathcal{J}}$ and $\overset{\checkmark}{\mathcal{J}}$. Note that Tapenade exhibits $O(n)$ space and time usage for all cases, while VLAD exhibits $O(n)$ space and time usage with $\overleftarrow{\mathcal{J}}$, but $O(1)$ space usage and $O(n)$ time usage with $\overset{\checkmark}{\mathcal{J}}$. The space complexity of $\overset{\checkmark}{\mathcal{J}}$ is the sum of the space required for the checkpoints and the space required for the tape. For a general computation of length t and maximal live storage w, the former is $O(w \log t)$ while the latter is $O(w)$. For the code in our example, $t = O(n)$ and $w = O(1)$, leading to the former being $O(\log n)$ and the latter being $O(1)$. We observe $O(1)$ space usage since the constant factors of the latter overpower the former. The time complexity of $\overset{\checkmark}{\mathcal{J}}$ is the sum of the time required to (re)compute the primal and the time required to perform the reverse sweep. For a general computation, the former is $O(t \log t)$ while the latter is $O(t)$. For the code in our example, the former is $O(n \log n)$ and the latter is $O(n)$. We observe $O(n)$ time usage since, again, the constant factors of the latter overpower the former.

Other methods present themselves for implementing the general-purpose checkpointing interface. One can use POSIX `fork()` much in the same way that it has been used to implement the requisite nondeterminism in probabilistic programming languages like probabilistic C [8]. A copy-on-write implementation of `fork()`, as is typical, would make this reasonably efficient and allow it to apply in the host, rather than the target, and thus could be used to provide an overloaded implementation of binomial checkpointing in a fashion that was largely transparent to the user. Alternatively, direct-style code could be compiled into CPS using a CPS transformation. A compiler for a language like VLAD can be constructed that generates target code in CPS that is instrumented with step counting, step limits, and checkpointing interruptions. A driver can be wrapped around such code to implement $\overset{\checkmark}{\mathcal{J}}$. Existing high-performance compilers, like SML/NJ [9], for functional languages like SML, already generate target code in CPS, so by adapting such to the purpose of AD with binomial checkpointing, it seems feasible to achieve high performance. In fact, the overhead of the requisite instrumentation for step counting, step limits, and checkpointing interruptions need not be onerous because the step counting, step limits, and checkpointing interruptions for basic blocks can be factored, and those for loops can be hoisted, much as is done for the instrumentation needed to support storage allocation and garbage collection in implementations like MLton [10], for languages like SML, that achieve very low overhead for automatic



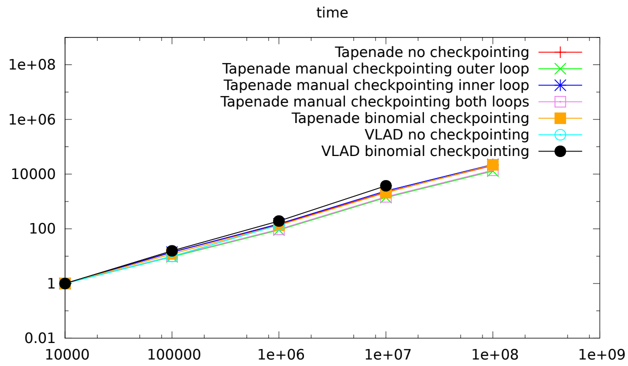


Figure 5: Space and time usage of reverse-mode AD with various checkpointing strategies, relative to the space and time for the first datapoint for each respective strategy.

storage management.

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] Yu. M. Volin and G. M. Ostrovskii. Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis. *Computers and Mathematics with Applications*, 11:1099–1114, 1985. doi: 10.1016/0898-1221(85)90188-9.

[2] Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 566–573, Heidelberg, 2006. Springer. ISBN 3-540-34385-7. doi: 10.1007/11758549_78.

[3] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[4] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, mar 2000. ISSN 0098-3500. doi: 10.1145/347837.347846. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[5] John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[6] C. Bendtsen and Ole 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.

[7] Gerald Jay Sussman and Guy L. Steele, Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, MIT, December 1975.

[8] Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. In *Proceedings of The 31st International Conference on Machine Learning*, pages 1935–1943, 2014.

[9] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[10] Stephen Weeks. Whole-program compilation in MLton, 2006. URL http://www.mlton.org/References.attachments/060916-mlton.pdf. Workshop on ML.