Divide-and-conquer checkpointing for arbitrary programs with no user annotation

Jeffrey Mark Siskind & Barak A. Pearlmutter

OPTIMIZATION METHODS & SOFTWARE
2018, VOL. 33, NOS. 4–6, 1288–1330
https://doi.org/10.1080/10556788.2018.1459621

Taylor & Francis
Taylor & Francis Group

OPEN ACCESS   Check for updates

Divide-and-conquer checkpointing for arbitrary programs with no user annotation

Jeffrey Mark Siskind [a] and Barak A. Pearlmutter [b]

[a]School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA; [b]Department of Computer Science, Maynooth University, Maynooth, Ireland

ABSTRACT
Classical reverse-mode automatic differentiation (AD) imposes only a small constant-factor overhead in operation count over the original computation, but has storage requirements that grow, in the worst case, in proportion to the time consumed by the original computation. This storage blowup can be ameliorated by checkpointing, a process that reorders application of classical reverse-mode AD over an execution interval to tradeoff space vs. time. Application of checkpointing in a divide-and-conquer fashion to strategically chosen nested execution intervals can break classical reverse-mode AD into stages which can reduce the worst-case growth in storage from linear to sublinear. Doing this has been fully automated only for computations of particularly simple form, with checkpoints spanning execution intervals resulting from a limited set of program constructs. Here we show how the technique can be automated for arbitrary computations. The essential innovation is to apply the technique at the level of the language implementation itself, thus allowing checkpoints to span any execution interval.

ARTICLE HISTORY
Received 19 March 2017
Accepted 12 March 2018

KEYWORDS
Reverse-mode automatic differentiation; binomial checkpointing; treeverse; programming language theory; compiler theory; lambda calculus

AMS SUBJECT CLASSIFICATIONS
68N20; 68N18; 65F50; 65D25; 46G05; 58C20

1. Introduction

Reverse-mode automatic differentiation (AD) traverses the run-time dataflow graph of a calculation in reverse order, in a so-called *reverse sweep*, so as to calculate a Jacobian-transpose-vector product of the Jacobian of the given original (or *primal*) calculation [22]. Although the number of arithmetic operations involved in this process is only a constant factor greater than that of the primal calculation, some values involved in the primal dataflow graph must be saved for use in the reverse sweep, thus imposing considerable storage overhead. This is accomplished by replacing the primal computation with a *forward sweep* that performs the primal computation while saving the requisite values on a data structure known as the *tape*. A technique called *checkpointing* [25] reorders portions of the forward and reverse sweeps to reduce the maximal length of the requisite tape. Doing so, however, requires (re)computation of portions of the primal and saving the requisite program state to support such as *snapshots*. Overall space savings result when the space

CONTACT Jeffrey Mark Siskind qobi@purdue.edu
This article was originally published with errors, which have now been corrected in the online version. Please see Correction https://doi.org/10.1080/10556788.2018.1527890

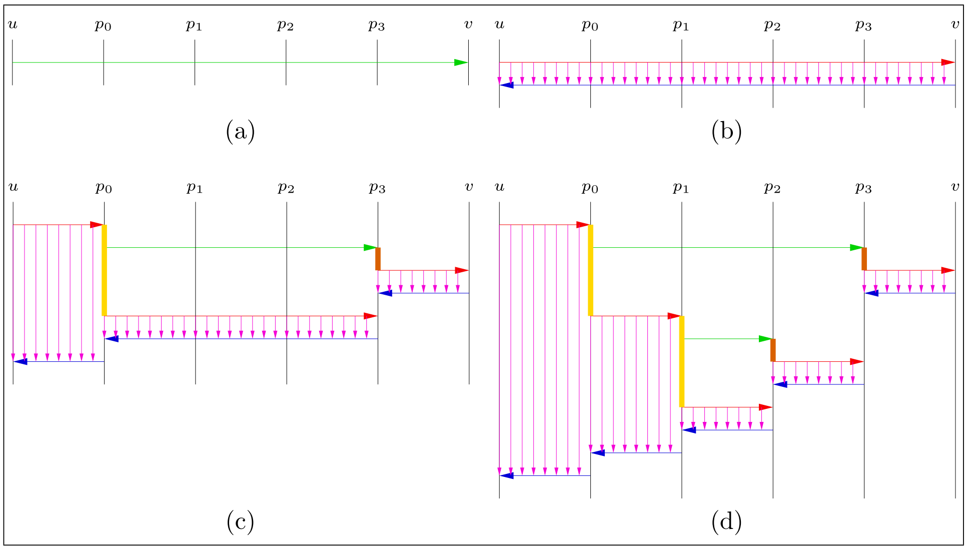

Figure 1. Checkpointing in reverse-mode AD. See text for description.

saved by reducing the maximal length of the requisite tape exceeds the space cost of storing the snapshots. Such space saving incurs a time cost in (re)computation of portions of the primal. Different checkpointing strategies lead to a space-time tradeoff.

We introduce some terminology that will be useful in describing checkpointing. An *execution point* is a point in time during the execution of a program. A *program point* is a location in the program code. Since program fragments might be *invoked* zero or more times during the execution of a program, each execution point corresponds to exactly one program point but each program point may correspond to zero or more execution points. An *execution interval* is a time interval spanning two execution points. A *program interval* is a fragment of code spanning two program points. Program intervals are usually constrained so that they nest, i.e. they do not cross one boundary of a syntactic program *construct* without crossing the other. Each program interval may correspond to zero or more execution intervals, those execution intervals whose endpoints result from the same invocation of the program interval. Each execution interval corresponds to at most one program interval. An execution interval might not correspond to a program interval because the endpoints might not result from the same invocation of any program interval.

Figures 1 and 2 illustrate the process of performing reverse-mode AD with and without checkpointing. Control flows from top to bottom, and along the direction of the arrow within each row. The symbols u, v, and $p_0, \dots, p_6$ denote execution points in the primal, u being the start of the computation whose derivative is desired, v being the end of that computation, and each p_i being an intermediate execution point in that computation. Reverse mode involves various sweeps, whose execution intervals are represented as horizontal green, red, and blue lines. Green lines denote (re)computation of the primal without taping. Red lines denote computation of the primal with taping, i.e. the forward sweep of reverse mode. Blue lines denote computation of the Jacobian-transpose-vector product, i.e. the reverse sweep of reverse mode. The vertical black lines denote collections of

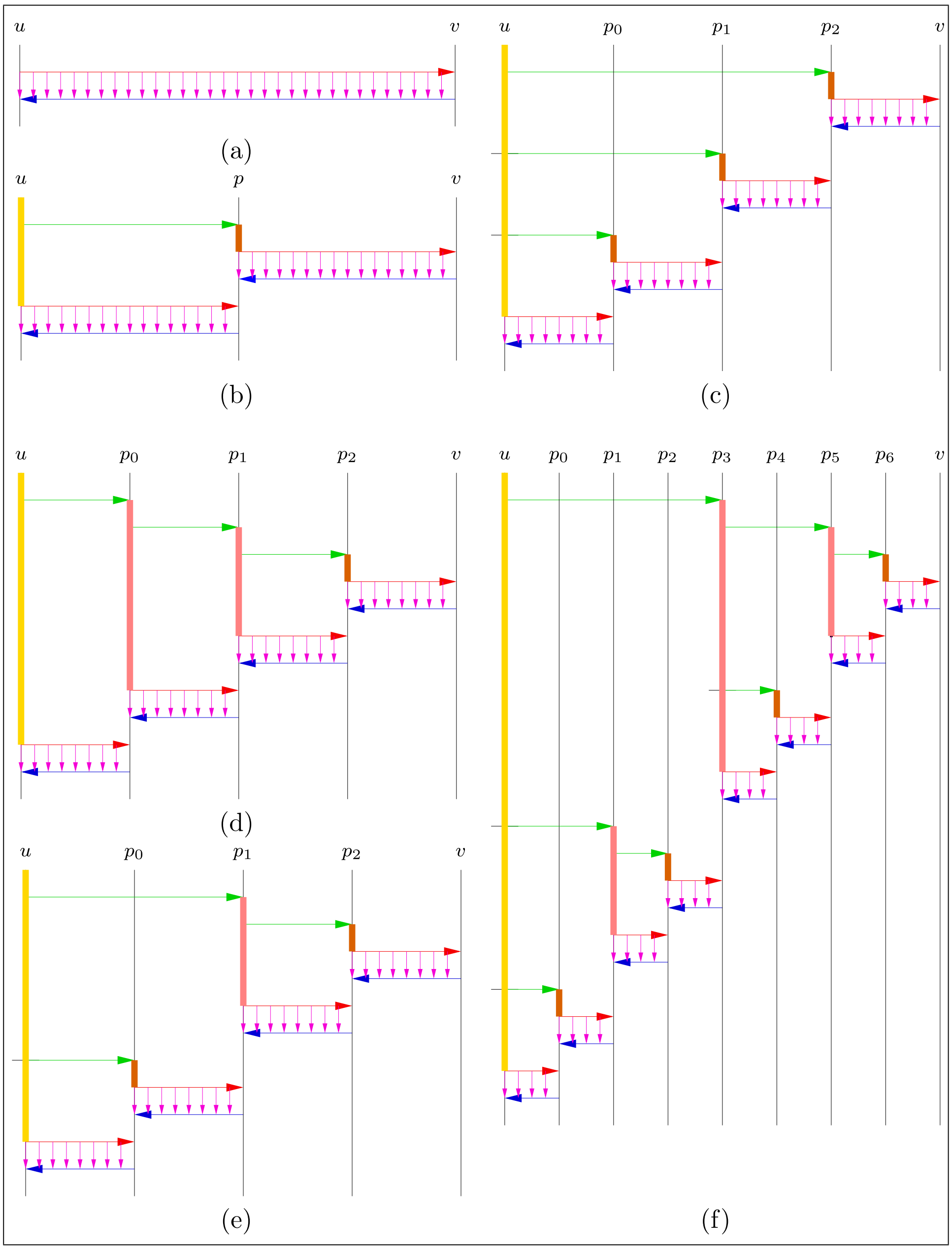

Figure 2. Divide-and-conquer checkpointing in reverse-mode AD. See text for description.

execution points across the various sweeps that correspond to execution points in the primal, each particular execution point being the intersection of a horizontal line and a vertical line. In portions of Figures 1 and 2 other than Figure 1(a) we refer to execution points for other sweeps besides the primal in a given collection with the symbols u, v, and $p_0, \ldots, p_6$ when the intent is clear. The vertical violet, gold, pink, and brown lines denote execution intervals for the lifetimes of various saved values. Violet lines denote the lifetime of a value saved on the tape during the forward sweep and used during the reverse sweep. The value is saved at the execution point at the top of the violet line and used once at the execution point at the bottom of that line. Gold and pink lines denote the lifetime of a snapshot.[1] The snapshot is saved at the execution point at the top of each gold or pink line and used at various other execution points during its lifetime. Green lines emanating from a gold or pink line indicate restarting a portion of the primal computation from a saved snapshot.

Figure 1(a) depicts the primal computation, $y = f(x)$, which takes t time steps, with x being a portion of the program state at execution point u and y being a portion of the program state at execution point v computed from x. This is performed without taping (green). Figure 1(b) depicts classical reverse mode without checkpointing. An uninterrupted forward sweep (red) is performed for the entire length of the primal, then an uninterrupted reverse sweep (blue) is performed for the entire length. Since the tape values are consumed in reverse order from which they are saved, the requisite tape length is $O(t)$. Figure 1(c) depicts a *checkpoint* introduced for the execution interval $[p_0, p_3)$. This interrupts the forward sweep and delays a portion of that sweep until the reverse sweep. Execution proceeds by a forward sweep (red) that tapes during the execution interval $[u, p_0)$, a primal sweep (green) without taping during the execution interval $[p_0, p_3)$, a taping forward sweep (red) during the execution interval $[p_3, v)$, a reverse sweep (blue) during the execution interval $[v, p_3)$, a taping forward sweep (red) during the execution interval $[p_0, p_3)$, a reverse sweep (blue) during the execution interval $[p_3, p_0)$, and then a reverse sweep (blue) during the execution interval $[p_0, u)$. The forward sweep for the execution interval $[p_0, p_3)$ is delayed until after the reverse sweep for the execution interval $[v, p_3)$. As a result of this reordering, the tapes required for those sweeps are not simultaneously live. Thus the requisite tape length is the maximum of the two tape lengths, not their sum. This savings comes at a cost. To allow such out-of-order execution, a snapshot (gold) must be saved at p_0 and the portion of the primal during the execution interval $[p_0, p_3)$ must be computed twice, first without taping (green) then with (red).

A checkpoint can be introduced into a portion of the forward sweep that has been delayed, as shown in Figure 1(d). An additional checkpoint can be introduced for the execution interval $[p_1, p_2)$. This will delay a portion of the already delayed forward sweep even further. As a result, the portions of the tape needed for the three execution intervals $[p_1, p_2)$, $[p_2, p_3)$, and $[p_3, v)$ are not simultaneously live, thus further reducing the requisite tape length, but requiring more (re)computation of the primal (green). The execution intervals for multiple checkpoints must either be disjoint or must nest; the execution interval of one checkpoint cannot cross one endpoint of the execution interval of another checkpoint without crossing the other endpoint.

Execution intervals for checkpoints can be specified in a variety of ways.

program interval
Execution intervals of specified program intervals constitute checkpoints.
subroutine call site
Execution intervals of specified subroutine call sites constitute checkpoints.
subroutine body
Execution intervals of specified subroutine bodies constitute checkpoints [25].

Nominally, these have the same power; with any one, one could achieve the effect of the other two. Specifying a subroutine body could be accomplished by specifying all call sites

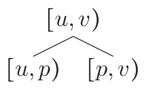

Figure 3. A binary checkpoint tree.

to that subroutine. Specifying some call sites but not others could be accomplished by having two variants of the subroutine, one whose body is specified and one whose is not, and calling the appropriate one at each call site. Specifying a program interval could be accomplished by extracting that interval as a subroutine.

Tapenade [12] allows the user to specify program intervals for checkpoints with the `c$ad checkpoint-start` and `c$ad checkpoint-end` pragmas. Tapenade, by default, checkpoints all subroutine calls [8]. This default can be overridden for named subroutines with the `-nocheckpoint` command-line option and for both named subroutines and specific call sites with the `c$ad nocheckpoint` pragma.

Recursive application of checkpointing in a divide-and-conquer fashion, i.e. ‘treeverse’, can divide the forward and reverse sweeps into stages run sequentially [9]. The key idea is that only one stage is live at a time, thus requiring a shorter tape. However, the state of the primal computation at various intermediate execution points needs to be saved as snapshots, in order to (re)run the requisite portion of the primal to allow the forward and reverse sweeps for each stage to run in turn. This process is illustrated in Figure 2. Consider a *root execution interval* $[u, v)$ of the derivative calculation. Without checkpointing, the forward and reverse sweeps span the entire root execution interval, as shown in Figure 2(a). One can divide the root execution interval $[u, v)$ into two subintervals $[u, p)$ and $[p, v)$ at the *split point* p and checkpoint the first subinterval $[u, p)$. This divides the forward (red) and reverse (blue) sweeps into two *stages*. These two stages are not simultaneously live. If the two subintervals are the same length, this halves the storage needed for the tape at the expense of running the primal computation for $[u, p)$ twice, first without taping (green), then with taping (red). This requires a single snapshot (gold) at u. This process can be viewed as constructing a *binary checkpoint tree* (Figure 3) whose nodes are labelled with execution intervals, the intervals of the children of a node are adjacent, the interval of a node is the disjoint union of the intervals of its children, and left children are checkpointed.

One can construct a left-branching binary checkpoint tree over the same root execution interval $[u, v)$ with the split points p_0, p_1, and p_2 (Figure 4(a)). This can also be viewed as constructing an *n-ary checkpoint tree* where all children but the rightmost are checkpointed (Figure 4(b)). This leads to *nested* checkpoints for the execution intervals $[u, p_0)$, $[u, p_1)$, and $[u, p_2)$ as shown in Figure 2(c). Since the starting execution point u is the same for these intervals, a single snapshot (gold) with longer lifetime suffices. These checkpoints divide the forward (red) and reverse (blue) sweeps into four stages. This allows the storage needed for the tape to be reduced arbitrarily (i.e. the red and blue segments can be made arbitrarily short), by rerunning successively shorter prefixes of the primal computation (green), without taping, running only short segments (red) with taping. This requires an $O(t)$ increase in time for (re)computation of the primal (green).

Alternatively, one can construct a right-branching binary checkpoint tree over the same root execution interval $[u, v)$ with the same split points p_0, p_1, and p_2 (Figure 5). This also divides the forward (red) and reverse (blue) sweeps into four stages. With this, the requisite tape length (the maximal length of the red and blue segments) can be reduced

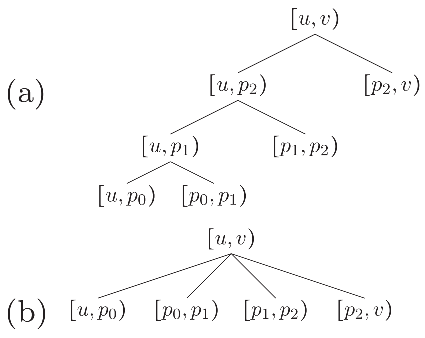

Figure 4. (a) a left-branching binary checkpoint tree and (b) equivalent n-ary checkpoint tree.

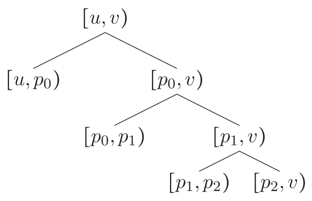

Figure 5. A right-branching checkpoint tree.

arbitrarily while running the primal (green) just once, by saving more snapshots (gold and pink), as shown in Figure 2(d), This requires an $O(t)$ increase in space for storage of the live snapshots (gold and pink).

Thus we see that divide-and-conquer checkpointing can make the requisite tape arbitrarily small with either left- or right-branching binary checkpoint trees. This involves a space-time tradeoff. The left-branching binary checkpoint trees require a single snapshot but an $O(t)$ increase in time for (re)computation of the primal (green). The right-branching binary checkpoint trees require an $O(t)$ increase in space for storage of the live snapshots (gold and pink) but (re)run the primal only once.

One can also construct a complete binary checkpoint tree over the same root execution interval $[u, v)$ with the same split points p_0, p_1, and p_2 (Figure 6). This constitutes application of the approach from Figure 2(b) in a divide-and-conquer fashion as shown in Figure 2(e). This also divides the forward (red) and reverse (blue) sweeps into four stages. One can continue this divide-and-conquer process further, with more split points, more snapshots, and more but shorter stages, as shown in Figure 2(f). This leads to an $O(\log t)$ increase in space for storage of the live snapshots (gold and pink) and an $O(\log t)$ increase in time for (re)computation of the primal (green). Variations of this technique can tradeoff between different improvements in space and/or time complexity, leading to overhead in a variety of sublinear asymptotic complexity classes in one or both. In order to apply this technique, we must be able to construct a checkpoint tree of the desired shape with appropriate split points. This in turn requires the ability to interrupt the primal computation at appropriate execution points, save the interrupted execution state as a *capsule*, and restart the computation from the capsules, sometimes repeatedly.[2]

Any given divide-and-conquer decomposition of the same root execution interval with the same split points can be viewed as either a binary checkpoint tree or an n-ary

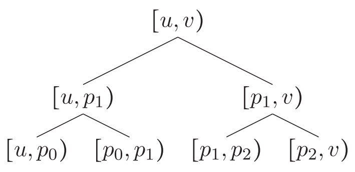

Figure 6. A complete binary checkpoint tree.

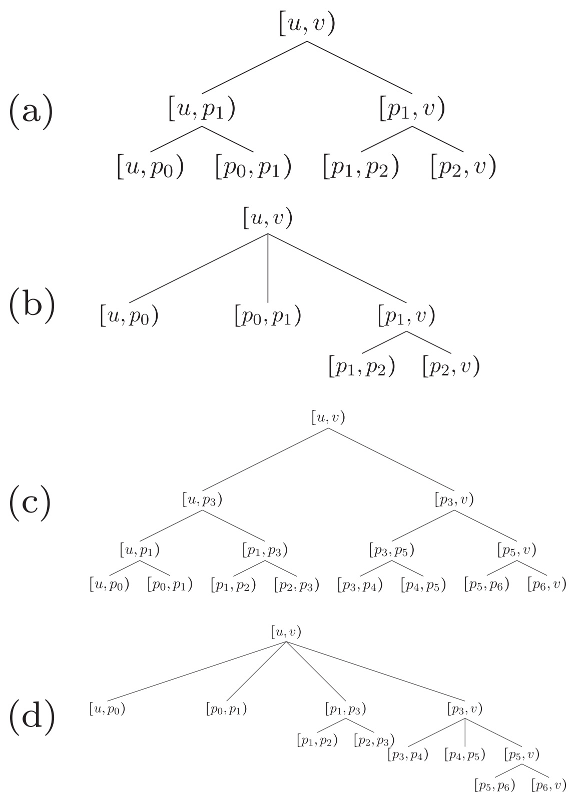

Figure 7. Figure 2(e) interpreted as (a) a binary checkpoint tree and as (b) an *n*-ary checkpoint tree. Figure 2(f) interpreted as (c) a binary checkpoint tree and as (d) an *n*-ary checkpoint tree.

checkpoint tree. Thus Figure 2(e) can be viewed as either Figure 7(a) or Figure 7(b). Similarly, Figure 2(f) can be viewed as either Figure 7(c) or Figure 7(d). Thus we distinguish between two algorithms to perform divide-and-conquer checkpointing.

binary An algorithm that constructs a binary checkpoint tree.
treeverse The algorithm from [9, Figures 2 and 3] that constructs an *n*-ary checkpoint tree.

There is, however, a simple correspondence between associated binary and *n*-ary checkpoint trees. The *n*-ary checkpoint tree is derived from the binary checkpoint tree by coalescing each maximal sequence of left branches into a single node. Thus we will see, in Section 5, that these two algorithms exhibit the same properties.

Note that (divide-and-conquer) checkpointing does not incur any space or time overhead in the forward or reverse sweeps themselves (i.e. the number of violet lines and the total length of red and blue lines). Any space overhead results from the snapshots (gold and pink) and any time overhead results from (re)computation of the primal (green).

Several design choices arise in the application of divide-and-conquer checkpointing in addition to the choice of binary vs. n-ary checkpoint trees.

- What root execution interval(s) should be subject to divide-and-conquer checkpointing?
- Which execution points are candidate split points? The divide-and-conquer process of constructing the checkpoint tree will select actual split points from these candidates.
- What is the shape or depth of the checkpoint tree, i.e. what is the termination criterion for the divide-and-conquer process?

Since the leaf nodes of the checkpoint tree correspond to stages, the termination criterion and the number of evaluation steps in the stage at each leaf node (the length of a pair of red and blue lines) are mutually constrained. The number of live snapshots at a leaf (how many gold and pink lines are crossed by a horizontal line drawn leftward from that stage, the pair of red and blue lines, to the root) depends on the depth of the leaf and its position in the checkpoint tree. Different checkpoint trees, with different shapes resulting from different termination criteria and split points, can lead to a different maximal number of live snapshots, resulting in different storage requirements. The amount of (re)computation of the primal (the total length of the green lines) can also depend on the shape of the checkpoint tree, thus different checkpoint trees, with different shapes resulting from different termination criteria and split points, can lead to different compute-time requirements. Thus different strategies for specifying the termination criterion and the split points can influence the space-time tradeoff.

We make a distinction between several different approaches to selecting root execution intervals subject to divide-and-conquer checkpointing.

loop Execution intervals resulting from invocations of specified DO loops are subject to divide-and-conquer checkpointing.

entire derivative calculation The execution interval for an entire specified derivative calculation is subject to divide-and-conquer checkpointing.

We further make a distinction between several different approaches to selecting candidate split points.

iteration boundary Iteration boundaries of the DO loop specified as the root execution interval are taken as candidate split points.

arbitrary Any execution point inside the root execution interval can be taken as a candidate split point.

We further make a distinction between several different approaches to specifying the termination criterion and deciding which candidate split points to select as actual split points.

bisection Split points are selected so as to divide the computation dominated by a node in half as one progresses successively from right to left among children [9, Equation (12)]. One can employ a variety of termination criteria, including that from [9, p. 46]. If the termination criterion is such that the total number of leaves is a power of two, one obtains a complete binary checkpoint tree. A termination criterion that bounds the number of evaluation steps in a leaf limits the size of the tape and achieves logarithmic overhead in both asymptotic space and time complexity compared with the primal.

binomial Split points are selected using the criterion from [9, Equation (16)]. The termination criterion from [9, p. 46] is usually adopted to achieve the desired properties discussed in [9]. Different termination criteria can be selected to control space-time tradeoffs.

- *fixed space overhead* One can bound the size of the tape and the number of snapshots to obtain sublinear but superlogarithmic overhead in asymptotic time complexity compared with the primal.
- *fixed time overhead* One can bound the size of the tape and the (re)computation of the primal to obtain sublinear but superlogarithmic overhead in asymptotic space complexity compared with the primal.
- *logarithmic space and time overhead* One can bound the size of the tape and obtain logarithmic overhead in both asymptotic space and time complexity compared with the primal. The constant factor is less than that of bisection checkpointing.

We elaborate on the strategies for selecting actual split points from candidate split points and the associated termination criteria in Section 5.

Divide-and-conquer checkpointing has only been provided to date in AD systems in special cases. For example, TAPENADE allows the user to select invocations of a specified DO loop as the root execution interval for divide-and-conquer checkpointing with the `c$ad binomial-ckp` pragma, taking iteration boundaries of that loop as candidate split points. TAPENADE employs binomial selection of split points and a fixed space overhead termination criterion. Note, however, that TAPENADE only guarantees this fixed space overhead property for DO loop bodies that take constant time. Similarly ADOL-C [10] contains a nested taping mechanism for time-integration processes [17] that also performs divide-and-conquer checkpointing. This only applies to code formulated as a time-integration process.

Here, we present a framework for applying divide-and-conquer checkpointing to arbitrary code with no special annotation or refactoring required. An entire specified derivative calculation is taken as the root execution interval, rather than invocations of a specified DO loop. Arbitrary execution points are taken as candidate split points, rather than iteration boundaries. As discussed below in Section 5, both binary and n-ary (treeverse) checkpoint trees are supported. Furthermore, as discussed below in Section 5, both bisection and binomial checkpointing are supported. Additionally, all of the above termination criteria are supported: fixed space overhead, fixed time overhead, and logarithmic space and time overhead. Any combination of the above checkpoint-tree generation algorithms, split-point selection methods, and termination criteria are supported. In order to apply this framework, we must be able to interrupt the primal computation at appropriate execution points, save the interrupted execution state as a capsule, and restart the computation from the capsules, sometimes repeatedly. This is accomplished by building divide-and-conquer checkpointing on top of a general-purpose mechanism for interrupting and resuming computation. This mechanism is similar to *engines* [13] and is orthogonal to AD. We present several implementations of our framework which we call CHECKPOINTVLAD. In Section 6, we compare the space and time usage of our framework with that of TAPENADE on an example.

Note that one cannot generally achieve the space and time guarantees of divide-and-conquer checkpointing with program-interval, subroutine-call-site, or subroutine-body checkpointing unless the call tree has the same shape as the requisite checkpoint tree. Furthermore, one cannot generally achieve the space and time guarantees of divide-and-conquer checkpointing for DO loops by specifying the loop body as a program-interval checkpoint, as that would lead to a right-branching checkpoint tree and behaviour analogous to Figure 2(d). Moreover, if one allows split points at arbitrary execution points, the resulting checkpoint execution intervals may not correspond to program intervals.

Some form of divide-and-conquer checkpointing is *necessary*. One may wish to take the gradient of a long-running computation, even if it has low asymptotic time complexity. The length of the tape required by reverse mode without divide-and-conquer checkpointing increases with increasing run time. Modern computers can execute several billion floating point operations per second, even without GPUs and multiple cores, which only exacerbate the problem. If each such operation required storage of a single eight-byte double precision number, modern terabyte RAM sizes would fill up after a few seconds of computation. Thus without some form of divide-and-conquer checkpointing, it would not be possible to efficiently take the gradient of a computation that takes more than a few seconds.

Machine learning methods in general, and deep learning methods in particular, require taking gradients of long-running high-dimension computations, particularly when training deep neural networks in general or recurrent neural networks over long time series. Thus variants of divide-and-conquer checkpointing have been rediscovered and deployed by the machine learning community in this context [6,11]. These implementations are far from automatic, and depend on compile-time analysis of the static primal flow graphs.

The general strategy of divide-and-conquer checkpointing, the n-ary treeverse algorithm, the bisection and binomial strategies for selecting split points, and the termination criteria that provide fixed space overhead, fixed time overhead, and logarithmic space and time overhead were all presented in [9]. Furthermore, TAPENADE has implemented divide-and-conquer checkpointing with the n-ary treeverse algorithm, the binomial strategy for selecting split points, and the termination criterion that provides fixed space overhead, but only for root execution intervals corresponding to invocations of specified DO loops that meet certain criteria with split points restricted to iteration boundaries of those loops. To our knowledge, the binary checkpoint-tree algorithm presented here and the framework for allowing it to achieve all of the same guarantees as the n-ary treeverse algorithm is new. However, our central novel contribution here is providing a framework for supporting either the binary checkpoint-tree algorithm or the n-ary treeverse algorithm, either bisection or binomial split point selection, and any of the termination criteria of fixed space overhead, fixed time overhead, or logarithmic space and time overhead in a way that supports taking the entire derivative calculation as the root execution interval and taking arbitrary execution points as candidate split points, by integrating the framework into the language implementation.

Some earlier work [14,15,23] prophetically presaged the work here. This work seems to have received far less exposure and attention than deserved. Perhaps because the ideas therein were so advanced and intricate that it was difficult to communicate those ideas clearly. Moreover, the authors report difficulties in getting their implementations to be fully functional. Our work here formulates the requisite ideas and mechanisms carefully and precisely, using methods from the programming-language community, like formulation of divide-and-conquer checkpointing of a function as divide-and-conquer application of reverse mode to two functions whose composition is the original function, formulation of the requisite decomposition as a precise and abstract interruption and resumption interface, formulation of semantics precisely through specification of evaluators, use of CPS evaluators to specify an implementation of the interruption and resumption interface, and systematic derivation of a compiler from that evaluator via CPS conversion, to allow complete, correct, comprehensible, and fully general implementation.

2. The limitations of divide-and-conquer checkpointing with split points at fixed syntactic program points like loop iteration boundaries

Consider the example in Figure 8. This example, $y = f(x; l, \phi)$, while contrived, is a simple caricature of a situation that arises commonly in practice: modelling a physical system with an adaptive grid. An initial state vector $x : \mathbb{R}^n$ is repeatedly transformed by a state update process $\mathbb{R}^n \rightarrow \mathbb{R}^n$ and, upon termination, an aggregate property y of the final state is computed by a function $\mathbb{R}^n \rightarrow \mathbb{R}$. We wish to compute the gradient of that property y relative to the initial state x. Here, the state update process first rotates the value pairs at adjacent odd-even coordinates of the state x by an angle θ and then rotates those at adjacent even-odd coordinates. The rotation θ is taken to be proportional to the magnitude of x. The adaptive grid manifests in two nested update loops. The outer loop has duration l, specified as an input hyperparameter. The duration m of the inner loop varies wildly as some function of another input hyperparameter ϕ and the outer loop index i, perhaps $2^{\lfloor \lg(l)\rfloor - \lfloor \lg(1+(1013\lfloor 3^x\rfloor i \mod l))\rfloor}$, that is small on most iterations of the outer loop but $O(l)$ on a few iterations. If the split points were limited to iteration boundaries of the outer loop, as would be common in existing implementations, the increase in space or time requirements would grow larger than sublinearly. The issue is that for the desired sublinear growth properties to hold, it must be possible to select arbitrary execution points as split points. In other words, the granularity of the divide-and-conquer decomposition must be primitive atomic computations, not loop iterations. The distribution of run time across the program is not modularly reflected in the static syntactic structure of the source code, in this case the loop structure. Often the user is unaware of, 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 it. Yet the user may still wish to reap the sublinear space or time overhead benefits of divide-and-conquer checkpointing. 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 will

```
      function ilog2(l)
      ilog2 = dlog(real(l, 8))/dlog(2.0d0)
      end

      subroutine rotate(theta, x1, x2, x1p, x2p)
      double precision theta, x1, x2, x1p, x2p, c, s
      c = cos(theta)
      s = sin(theta)
      x1p = c*x1-s*x2
      x2p = s*x1+c*x2
      end

      subroutine rot1(theta, n, x)
      double precision theta, x, x1p, x2p
      dimension x(n)
      do k = 1, n-1, 2
         call rotate(theta, x(k), x(k+1), x1p, x2p)
         x(k) = x1p
         x(k+1) = x2p
      end do
      end

      subroutine rot2(theta, n, x)
      double precision theta, x, x1p, x2p
      dimension x(n)
      do k = 2, n-2, 2
         call rotate(theta, x(k), x(k+1), x1p, x2p)
         x(k) = x1p
         x(k+1) = x2p
      end do
      end

      subroutine magsqr(n, x, y)
      double precision x, y
      dimension x(n)
      y = 0.0d0
      do k = 1, n
         y = y+x(k)*x(k)
      end do
      end
```

```
      subroutine f(n, x, l, phi, y)
      double precision phi, x, y, x1
      dimension x(n), x1(1000)
      do k = 1, n
         x1(k) = x(k)
      end do
c$ad binomial-ckp l+1 40 1
      do i = 1, l
         m = 2**(ilog2(l)-
     +           ilog2(1+int(mod(1013.0d0*3.0d0**phi*real(i, 8),
     +                               real(l, 8)))))
         do j = 1, m
            call magsqr(n, x1, y)
            y = sqrt(y)
            call rot1(1.2d0*y, n, x1)
            call rot2(1.4d0*y, n, x1)
         end do
      end do
      call magsqr(n, x1, y)
      y = y/2.0d0
      end

      program main
      double precision phi, x, xb, y, yb
      dimension x(1000), xb(1000)
      read *, n
      read *, l
      read *, phi
      do k = 1, n
         x(k) = n+1-k
         xb(k) = 0.0d0
      end do
      yb = 1.0d0
      call f(n, x, l, phi, y)
      print *, y
      call f_b(n, x, xb, l, phi, y, yb)
      do k = 1, n
         print *, xb(k)
      end do
      end
```

Figure 8. FORTRAN example. This example is rendered in CHECKPOINTVLAD in Figure 28. Space and time overhead of two variants of this example when run under TAPENADE are presented in Figure 29. The pragma used for the variant with divide-and-conquer checkpointing of the outer DO loop is shown. This pragma is removed for the variant with no checkpointing.

now proceed to discuss an implementation strategy for divide-and-conquer checkpointing that does not constrain split points to loop iteration boundaries or other syntactic program constructs and does not constrain checkpoints to program intervals or other syntactic program constructs. Instead, it can take any arbitrary execution point as a split point and introduce checkpoints at any resulting execution interval.

3. Technical details of our method

Implementing divide-and-conquer checkpointing requires the capacity to

(1) measure the length of the primal computation,
(2) interrupt the primal computation at a portion of the measured length,
(3) save the state of the interrupted computation as a capsule, and
(4) resume an interrupted computation from a capsule.

For our purposes, the second and third operations are always performed together and can be fused into a single operation. These can be difficult to implement efficiently as library routines in an existing language implementation (see Section 7.1). Thus we design a new language implementation, CHECKPOINTVLAD, with efficient support for these low-level operations.

3.1. Core language

CHECKPOINTVLAD adds builtin AD operators to a functional pre-AD core language. In the actual implementation, this core language is provided with a SCHEME-like surface syntax.[3]

But nothing turns on this; the core language can be exposed with any surface syntax. For expository purposes, we present the core language here in a simple, more traditional, math-like notation.

CHECKPOINTVLAD employs the same functional core language as our earlier VLAD system [21]. Support for AD in general, and divide-and-conquer checkpointing in particular, is simplified in a functional programming language (see Section 7.2). Except for this simplification, which can be eliminated with well-known techniques (e.g. monads [26] and uniqueness types [1]) for supporting mutation in functional languages, nothing turns on our choice of core language. We intend our core language as a simple expository vehicle for the ideas presented here; they could be implemented in other core languages (see Section 7.1).

Our core language contains the following constructs:

$$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \diamond e \mid e_1 \bullet e_2 \tag{1}$$

Here, e denotes expressions, c denotes constants, x denotes variables, $e_1\ e_2$ denotes function application, $\diamond$ denotes builtin unary operators, and $\bullet$ denotes builtin binary operators. For expository simplicity, the discussion of the core language here omits many vagaries such as support for recursion and functions of multiple arguments; the actual implementation supports these using standard mechanisms that are well known within the programming-language community (e.g. tupling or Currying).

3.2. Direct-style evaluator for the core language

We start by formulating a simple evaluator for this core language (Figure 9) and extend it to perform AD and ultimately divide-and-conquer checkpointing. 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’ and $\mathcal{A}$, denoting ‘apply’) take inputs as function-call arguments and yield outputs as function-call return values [18]. While this evaluator can be viewed as an interpreter, it is intended more as a description of the evaluation mechanism; this mechanism could be the underlying hardware as exposed via a compiler. Indeed, as described below in Section 3.14, we have written three implementations, one an interpreter, one a hybrid compiler/interpreter, and one a compiler.

$$\mathcal{A}\ \langle(\lambda x.e),\rho\rangle\ v = \mathcal{E}\ \rho[x\mapsto v]\ e \tag{2a}$$
$$\mathcal{E}\ \rho\ c = c \tag{2b}$$
$$\mathcal{E}\ \rho\ x = \rho\ x \tag{2c}$$
$$\mathcal{E}\ \rho\ (\lambda x.e) = \langle(\lambda x.e),\rho\rangle \tag{2d}$$
$$\mathcal{E}\ \rho\ (e_1\ e_2) = \mathcal{A}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2) \tag{2e}$$
$$\mathcal{E}\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \textbf{if}\ (\mathcal{E}\ \rho\ e_1)\ \textbf{then}\ (\mathcal{E}\ \rho\ e_2)\ \textbf{else}\ (\mathcal{E}\ \rho\ e_3) \tag{2f}$$
$$\mathcal{E}\ \rho\ (\diamond e) = \diamond(\mathcal{E}\ \rho\ e) \tag{2g}$$
$$\mathcal{E}\ \rho\ (e_1 \bullet e_2) = (\mathcal{E}\ \rho\ e_1) \bullet (\mathcal{E}\ \rho\ e_2) \tag{2h}$$

Figure 9. Direct-style evaluator for the core CHECKPOINTVLAD language.

With any evaluator, one distinguishes between two language evaluation strata: the *target*, the language being implemented and the process of evaluating programs in that language, and the *host*, the language in which the evaluator is written and the process of evaluating the evaluator itself. In our case, the target is CHECKPOINTVLAD, while the host varies among our three implementations; for the first two it is SCHEME while for the third it is the underlying hardware, achieved by compilation to machine code via C.

In the evaluator in Figure 9, ρ denotes an *environment*, a mapping from variables to their values, ρ_0 denotes the empty environment that does not map any variables, $\rho\ x$ denotes looking up the variable x in the environment ρ to obtain its value, $\rho[x \mapsto v]$ denotes augmenting an environment ρ to map the variable x to the value v, and $\mathcal{E}\ \rho\ e$ denotes evaluating the expression e in the context of the environment ρ. There is a clause for $\mathcal{E}$ in Figure 9, (2b–h), for each construct in (1). Clause (2b) says that one evaluates a constant by returning that constant. Clause (2c) says that one evaluates a variable by returning its value in the environment. The notation $\langle e, \rho\rangle$ denotes a *closure*, a lambda expression e together with an environment ρ containing values for the free variables in e. Clause (2d) says that one evaluates a lambda expression by returning a closure with the environment in the context that the lambda expression was evaluated in. Clause (2e) says that one evaluates an application by evaluating the callee expression to obtain a closure, evaluating the argument expression to obtain a value, and then applying the closure to the value with $\mathcal{A}$. $\mathcal{A}$, as described in (2a), evaluates the body of the lambda expression in the callee closure in the environment of that closure augmented with the formal parameter of that lambda expression bound to the argument value. The remaining clauses are all analogous to clause (2h), which says that one evaluates an expression $e_1 \bullet e_2$ in the target by evaluating e_1 and e_2 to obtain values and then applying $\bullet$ in the host to these values.

3.3. Adding AD operators to the core language

Unlike many AD systems implemented as libraries, we provide support for AD by augmenting the core language to include builtin AD operators for both forward and reverse mode [21]. This allows seamless integration of AD into the language in a completely general fashion with no unimplemented or erroneous corner cases. In particular, it allows nesting [19]. In CHECKPOINTVLAD, we adopt slight variants of the $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators previously incorporated into VLAD. (Nothing turns on this. The variants adopted here are simpler, better suit our expository purposes, and allow us to focus on the issue at hand.) In CHECKPOINTVLAD, these operators have the following signatures.

$$\overrightarrow{\mathcal{J}} : f\ x\ \acute{x} \mapsto (y, \acute{y}), \quad \overleftarrow{\mathcal{J}} : f\ x\ \grave{y} \mapsto (y, \grave{x}) \tag{3}$$

We use the notation $\acute{x}$ and $\grave{x}$ to denote tangent or cotangent values associated with the primal value x, respectively, and the notation (x, y) to denote a pair of values. Since in CHECKPOINTVLAD, functions can take multiple arguments but only return a single result, which can be an aggregate like a pair, the AD operators take the primal and the associated (co)tangent as distinct arguments but return the primal and the associated (co)tangent as a pair of values.

The $\overrightarrow{\mathcal{J}}$ operator provides the portal to forward mode and 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 provides the portal to reverse mode and calls a function f on a primal x with a cotangent $\grave{y}$ to yield a primal y and a cotangent $\grave{x}$.[4] Unlike the original VLAD, 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). Nothing turns on this; it allows us to focus on the issue at hand.

The implementations of VLAD and CHECKPOINTVLAD are disjoint and use completely different technology. The STALIN∇ [21] implementation of VLAD is based on source-code transformation, conceptually applied reflectively at run time but migrated to compile time through partial evaluation. The implementation of CHECKPOINTVLAD uses something more akin to operator overloading. Again, nothing turns on this; this simplification is for expository purposes and allows us to focus on the issue at hand (see Section 7.1).

In CHECKPOINTVLAD, AD is performed by overloading the arithmetic operations in the host, in a fashion similar to FADBAD++ [5]. The actual method used is that employed by R6RS-AD[5] and DIFFSHARP[6]. The key difference is that FADBAD++ uses C++ templates to encode a hierarchy of distinct forward-mode types (e.g. `F<double>`, `F<F<double> >`, `...`), distinct reverse-mode types (e.g. `B<double>`, `B<B<double> >`, `...`), and mixtures thereof (e.g. `F<B<double> >`,`B<F<double> >`, `...`) while here, we use a dynamic, run-time approach where numeric values are tagged with the nesting level [19,24]. Template instantiation at compile-time specializes code to different nesting levels. The dynamic approach allows a single interpreter (host), formulated around unspecialized code, to interpret different target programs with different nesting levels.

3.4. Augmenting the direct-style evaluator to support the AD operators

We add AD into the target language as new constructs.

$$e ::= \overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3 \mid \overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3 \tag{4}$$

We implement this functionality by augmenting the direct-style evaluator with new clauses for $\mathcal{E}$ (Figure 10), clause (2k) for $\overrightarrow{\mathcal{J}}$ and clause (2l) for $\overleftarrow{\mathcal{J}}$. These clauses are all analogous to clause (2h), formulated around $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators in the host. These are defined in (2i, 2j). The $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators in the host behave like $\mathcal{A}$ except that they level shift to perform AD. Just like $(\mathcal{A}\ f\ x)$ applies a target function f (closure) to a target value x, $(\overrightarrow{\mathcal{J}}\ f\ x\ \acute{x})$

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \textbf{let}\ (v_4 \triangleright \acute{v}_5) = (\mathcal{A}\ v_1\ (v_2 \triangleright \acute{v}_3))\ \textbf{in}\ (v_4, \acute{v}_5) \tag{2i}$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \textbf{let}\ (v_4 \triangleleft \grave{v}_5) = ((\mathcal{A}\ v_1\ v_2) \triangleleft \grave{v}_3)\ \textbf{in}\ (v_4, \grave{v}_5) \tag{2j}$$

$$\mathcal{E}\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \overrightarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3) \tag{2k}$$

$$\mathcal{E}\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \overleftarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3) \tag{2l}$$

Figure 10. Additions to the direct-style evaluator for CHECKPOINTVLAD to support AD.

performs forward mode by applying a target function f (closure) to a target primal value x and a target tangent value $\acute{x}$, while $(\overleftarrow{\mathcal{J}}\ f\ x\ \grave{y})$ performs reverse mode by applying a target function f (closure) to a target primal value x and a target cotangent value $\grave{y}$.

As described in (2i), $\overrightarrow{\mathcal{J}}$ operates by recursively walking v_2, a data structure containing primals, in tandem with $\acute{v}_3$, a data structure containing tangents, to yield a single data structure where each numeric leaf value is a dual number, a numeric primal value associated with a numeric tangent value. This recursive walk is denoted as $v_2 \rhd \acute{v}_3$. $\mathcal{A}$ is then used to apply the function (closure) v_1 to the data structure produced by $v_2 \rhd \acute{v}_3$. Since the input argument is level shifted and contains dual numbers instead of ordinary reals, the underlying arithmetic operators invoked during the application perform forward mode by dispatching on the tags at run time. The call to $\mathcal{A}$ yields a result data structure where each numeric leaf value is a dual number. This is then recursively walked to separate out two data structures, one, v_4, containing the numeric primal result values, and the other, $\acute{v}_5$, containing the numeric tangent result values, which are returned as a pair $(v_4, \acute{v}_5)$ This recursive walk is denoted as $\mathbf{let}\ (v_4 \rhd \acute{v}_5) = \ldots \mathbf{in}\ \ldots$.

As described in (2j), $\overleftarrow{\mathcal{J}}$ operates by recursively walking v_2, a data structure containing primals, to replace each numeric value with a tape node. $\mathcal{A}$ is then used to apply the function (closure) v_1 to this modified v_2. Since the input argument is level shifted and contains tape nodes instead of ordinary reals, the underlying arithmetic operators invoked during the application perform the forward sweep of reverse mode by dispatching on the tags at run time. The call to $\mathcal{A}$ yields a result data structure where each numeric leaf value is a tape node. A recursive walk is performed on this result data structure, in tandem with a data structure $\grave{v}_3$ of associated cotangent values, to initiate the reverse sweep of reverse mode. This combined operation is denoted as $((\mathcal{A}\ v_1\ v_2) \lhd \grave{v}_3)$. The result of the forward sweep is then recursively walked to replace each tape node with its numeric primal value and the input value is recursively walked to replace each tape node with the cotangent computed by the reverse sweep. These are returned as a pair $(v_4, \grave{v}_5)$. This combined operation is denoted as $\mathbf{let}\ (v_4 \lhd \grave{v}_5) = \ldots \mathbf{in}\ \ldots$.

3.5. An operator to perform divide-and-conquer checkpointing in reverse-mode AD

We introduce a new AD operator $\overset{\checkmark}{\mathcal{J}}$ to perform divide-and-conquer checkpointing.[7] (For expository simplicity, we focus for now on binary bisection checkpointing. In Section 5, we provide alternate implementations of $\overset{\checkmark}{\mathcal{J}}$ that perform treeverse and/or 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 various forms of divide-and-conquer checkpointing, save interchanging calls to $\overleftarrow{\mathcal{J}}$ and $\overset{\checkmark}{\mathcal{J}}$.

Conceptually, the behaviour of $\overset{\checkmark}{\mathcal{J}}$ is shown in Figure 11. In this inductive definition, a function f is split into the composition of two functions g and h in step (1), the capsule z

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

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

Figure 11. Algorithm for binary checkpointing.

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

Figure 12. General-purpose interruption and resumption interface.

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 behaviour 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 evaluation steps, and we terminate the recursion when f takes a bounded number of steps, the recursive divide-and-conquer process yields logarithmic asymptotic space/time overhead complexity.

The central difficulty in implementing the above is performing step (1), namely splitting a function f into two functions g and h, such that $f = h \circ g$, ideally where we can specify the split point, the number of evaluation steps through f where g transitions into h. A sophisticated user can manually rewrite a subprogram f into two subprograms g and h. A sufficiently powerful compiler or source transformation tool might also be able to do so, with access to non-local program text. But an overloading system, with access only to local information, would not be able to.

3.6. General-purpose interruption and resumption mechanism

We solve this problem by providing an interface to a general-purpose interruption and resumption mechanism that is orthogonal to AD (Figure 12). This interface allows (a) determining the number of evaluation 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 two in detail below, in Sections 3.8 and 3.12, and briefly discuss another in Section 7.1.

$$\begin{array}{lll}
\multicolumn{2}{l}{\text{To compute } (y,\grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}\text{:}} & \\
\textbf{base case:} & (y,\grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y} & \text{(step 0)} \\
\textbf{inductive case:} & l = \textsc{primops}\ f\ x & \text{(step 1)} \\
 & z = \textsc{interrupt}\ f\ x\ \lfloor \frac{l}{2} \rfloor & \text{(step 2)} \\
 & (y,\grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\textsc{resume}\ z)\ z\ \grave{y} & \text{(step 3)} \\
 & (z,\grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda x.\textsc{interrupt}\ f\ x\ \lfloor \frac{l}{2} \rfloor)\ x\ \grave{z} & \text{(step 4)}
\end{array}$$

Figure 13. Binary bisection checkpointing via the general-purpose interruption and resumption interface. Step (1) need only be performed once at the beginning of the recursion, with steps (2) and (4) taking l at the next recursion level to be $\lfloor l/2 \rfloor$ and step (3) taking l at the next recursion level to be $\lceil l/2 \rceil$. As discussed in the text, this implementation is not quite correct, because ($\lambda z.\textsc{resume}\ z$) in step (3) and ($\lambda x.\textsc{interrupt}\ f\ x\ \lfloor l/2 \rfloor$) in step (4) are host closures but need to be target closures. A proper implementation is given in Figure 18.

Irrespective of how one implements the general-purpose interruption and resumption interface, one can use it to implement the binary bisection variant of $\overset{\checkmark}{\mathcal{J}}$ in the host, as shown in Figure 13. The function f is split into the composition of two functions g and h by taking g as ($\lambda x.\textsc{interrupt}\ f\ x\ l$), where l is half the number of steps determined by ($\textsc{primops}\ f\ x$), and h as ($\lambda z.\textsc{resume}\ z$).

3.7. *Continuation-passing-style evaluator*

One way of implementing the general-purpose interruption and resumption interface is to convert the evaluator from direct style to what is known in the programming-language community as *continuation-passing style* (CPS) [18], 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 (Figures 14 and 15). In CPS, functions never return: they just call their continuation. With tail-call merging, this corresponds to a computed `go to` and does not incur stack growth. This crucially allows an interruption to actually return a capsule 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 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 an interrupt is triggered.

Figure 14 contains the portion of the CPS evaluator for the core language corresponding to Figure 9, while Figure 15 contains the portion of the CPS evaluator for the AD constructs corresponding to Figure 10. Except for (5b), the equations in Figures 9 and 10 are in one-to-one correspondence to those in Figures 14 and 15, in order. Clauses (5c–e) are analogous to the corresponding clauses (2b–d) except that they call the continuation k with the result, instead of returning that result. The remaining clauses for $\mathcal{E}$ in the CPS evaluator are all

$$\mathcal{A}\;k\;n\;l\;\langle(\lambda x.e),\rho\rangle\;v = \mathcal{E}\;k\;n\;l\;\rho[x\mapsto v]\;e \tag{5a}$$

$$\mathcal{E}\;k\;l\;l\;\rho\;e = [\![k,\langle(\lambda\_.e),\rho\rangle]\!] \tag{5b}$$

$$\mathcal{E}\;k\;n\;l\;\rho\;c = k\;(n+1)\;l\;c \tag{5c}$$

$$\mathcal{E}\;k\;n\;l\;\rho\;x = k\;(n+1)\;l\;(\rho\;x) \tag{5d}$$

$$\mathcal{E}\;k\;n\;l\;\rho\;(\lambda x.e) = k\;(n+1)\;l\;\langle(\lambda x.e),\rho\rangle \tag{5e}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(e_1\;e_2) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\&\qquad\quad(\mathcal{A}\;k\;n\;l\;v_1\;v_2))\\&\qquad\;n\;l\;\rho\;e_2))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{5f}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(\textbf{if}\;e_1\;\textbf{then}\;e_2\;\textbf{else}\;e_3) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\textbf{if}\;v_1\\&\quad\;\textbf{then}\;(\mathcal{E}\;k\;n\;l\;\rho\;e_2)\\&\quad\;\textbf{else}\;(\mathcal{E}\;k\;n\;l\;\rho\;e_3)))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{5g}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(\diamond e) = \mathcal{E}\;&(\lambda n\;l\;v.\\&\quad(k\;n\;l\;(\diamond v)))\\&(n+1)\;l\;\rho\;e\end{aligned} \tag{5h}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(e_1\bullet e_2) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\&\qquad\quad(k\;n\;l\;(v_1\bullet v_2)))\\&\qquad\;n\;l\;\rho\;e_2))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{5i}$$

Figure 14. CPS evaluator for the core CHECKPOINTVLAD language.

$$\begin{aligned}\overrightarrow{\mathcal{J}}\;v_1\;v_2\;\acute{v}_3 = \mathcal{A}\;&(\lambda n\;l\;(v_4\triangleright\acute{v}_5).\\&\quad(v_4,\acute{v}_5))\\&0\;\infty\;v_1\;(v_2\triangleright\acute{v}_3)\end{aligned} \tag{5j}$$

$$\begin{aligned}\overleftarrow{\mathcal{J}}\;v_1\;v_2\;\grave{v}_3 = \mathcal{A}\;&(\lambda n\;l\;v.\\&\quad\textbf{let}\;(v_4\triangleleft\grave{v}_5) = v\triangleleft\grave{v}_3\\&\quad\textbf{in}\;(v_4,\grave{v}_5))\\&0\;\infty\;v_1\;v_2\end{aligned} \tag{5k}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(\overrightarrow{\mathcal{J}}\;e_1\;e_2\;e_3) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\&\qquad\quad(\mathcal{E}\;(\lambda n\;l\;v_3.(k\;n\;l\;(\overrightarrow{\mathcal{J}}\;v_1\;v_2\;v_3)))\\&\qquad\qquad n\;l\;\rho\;e_3))\\&\qquad\;n\;l\;\rho\;e_2))\\&(n+1)\;l\;\;\rho\;e_1\end{aligned} \tag{5l}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(\overleftarrow{\mathcal{J}}\;e_1\;e_2\;e_3) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\&\qquad\quad(\mathcal{E}\;(\lambda n\;l\;v_3.(k\;n\;l\;(\overleftarrow{\mathcal{J}}\;v_1\;v_2\;v_3)))\\&\qquad\qquad n\;l\;\rho\;e_3))\\&\qquad\;n\;l\;\rho\;e_2))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{5m}$$

Figure 15. Additions to the CPS evaluator for CHECKPOINTVLAD to support AD.

variants of

$$\mathcal{E}\ (\lambda n\ l\ v_1.(\mathcal{E}\ (\lambda n\ l\ v_2.(k\ n\ l\ \ldots))\ n\ l\ \rho\ e_2))\ (n+1)\ l\ \rho\ e_1 \tag{6}$$

for one-, two-, or three-argument constructs. This evaluates the first argument e_1 and calls the continuation $(\lambda n\ l\ v_1.\ldots)$ with its value v_1. This continuation then evaluates the second argument e_2 and calls the continuation $(\lambda n\ l\ v_2.\ldots)$ with its value v_2. This continuation computes something, denoted by $\ldots$, and calls the continuation k with the resulting value.

The CPS evaluator threads a step count n and a step limit l through the evaluation process. Each clause of $\mathcal{E}$ increments the step count exactly once to provide a coherent fine-grained measurement of the execution time. Clause (5b) of $\mathcal{E}$ implements interruption. When the step count reaches the step limit, a capsule containing the saved state of the evaluator, denoted $[\![k,f]\!]$, is returned. Here, f is a closure $\langle(\lambda\_.e),\rho\rangle$ containing the environment ρ and the expression e at the time of interruption. This closure takes an argument that is not used. The step count n must equal the step limit l at the time of interruption. As will be discussed below, in Section 3.8, neither the step count nor the step limit need to be saved in the capsule, as the computation is always resumed with different step count and limit values.

Several things about this CPS evaluator are of note. First, all builtin unary and binary operators are assumed to take unit time. This follows from the fact that all clauses for $\mathcal{E}$, as typified by (6), increment the step count by one. Second, the builtin unary and binary operators in the host are implemented in direct style and are not passed a continuation. This means that clauses (5h, 5i), as typified by (6), must call the continuation k on the result of the unary and binary operators. Third, like all builtin operators, invocations of the $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators, including the application of v_1, are assumed to take unit time. This follows from the fact that clauses (5l, 5m), again as typified by (6), increment the step count by one. Fourth, like all builtin operators, $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ in the host, in (5j, 5k), are implemented in direct style and are not passed a continuation. This means that clauses (5l, 5m), as typified by (6), must call the continuation k on the result of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$. Finally, since $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ receive target functions (closures) for v_1, they must apply these to their arguments with $\mathcal{A}$. Since $\mathcal{A}$ is written in CPS in the CPS evaluator, these calls to $\mathcal{A}$ in (5j, 5k) must be provided with a continuation k, a step count n, and a step limit l as arguments. The continuation argument simply returns the result. The step count, however, is restarted at zero, and the step limit is set to ∞. This means that invocations of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ are atomic and cannot be interrupted internally. We discuss this further below in Section 7.3.

3.8. Implementing the general-purpose interruption and resumption interface with the CPS evaluator

With this CPS evaluator, it is possible to implement the general-purpose interruption and resumption interface (Figure 16). The implementation of PRIMOPS (7a) calls the evaluator with no step limit and simply counts the number of steps to completion. The implementation of INTERRUPT (7b) calls the evaluator with a step limit that must be smaller than that needed to complete so an interrupt is forced and the capsule $[\![k,\langle(\lambda\_.e),\rho\rangle]\!]$ is returned. The implementation of RESUME (7c) calls the evaluator with arguments from the saved

$$\textsc{primops}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x \tag{7a}$$
$$\textsc{interrupt}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x \tag{7b}$$
$$\textsc{resume}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot \tag{7c}$$

Figure 16. Implementation of the general-purpose interruption and resumption interface using the CPS evaluator.

capsule. Since the closure in the capsule does not use its argument, an arbitrary value $\bot$ is passed as that argument.

Note that the calls to $\mathcal{A}$ in $\overrightarrow{\mathcal{J}}$ (5j), $\overleftarrow{\mathcal{J}}$ (5k), PRIMOPS (7a), INTERRUPT (7b), and RESUME (7c) are the only portals into the CPS evaluator. The only additional call to $\mathcal{A}$ is in the evaluator itself, clause (5f) of $\mathcal{E}$. All of the portals restart the step count at zero. Except for the call in INTERRUPT (7b), none of the portals call the evaluator with a step limit. In particular, RESUME (7c) does not provide a step limit; other mechanisms detailed below provide for interrupting a resumed capsule.

This implementation of the general-purpose interruption and resumption interface cannot be used to fully implement $\overset{\checkmark}{\mathcal{J}}$ in the host as depicted in Figure 13. The reason is that the calls to $\overleftarrow{\mathcal{J}}$ in the base case, step (0), and INTERRUPT in step (2), must take a target function (closure) for *f*, because that is what is invoked by the calls to $\mathcal{A}$ in $\overleftarrow{\mathcal{J}}$ (5k) and INTERRUPT (7b). As written in Figure 13, the recursive calls to $\overset{\checkmark}{\mathcal{J}}$, namely steps (3) and (4), pass ($\lambda z.\textsc{resume}\ z$) and ($\lambda x.\textsc{interrupt}\ f\ x\ \lfloor l/2 \rfloor$) for *f*. There are two problems with this. First, these are host closures produced by host lambda expressions, not target closures. Second, these call the host functions RESUME and INTERRUPT that are not available in the target. Thus it is not possible to formulate these as target closures without additional machinery.

Examination of Figure 13 reveals that the general-purpose interruption and resumption interface is invoked four times in the implementation of $\overset{\checkmark}{\mathcal{J}}$. PRIMOPS is invoked in step (1), INTERRUPT is invoked in steps (2) and (4), and RESUME is invoked in step (3). Of these, PRIMOPS is invoked only in the host, RESUME is invoked only in the target, and INTERRUPT is invoked in both the host and the target. Thus we need to expose INTERRUPT and RESUME to the target. We do not need to expose PRIMOPS to the target; the implementation in Figure 13 only uses it in the host. For INTERRUPT, the call in step (2) can use the host implementation (7b) in Figure 16 but the call in step (4) must use a new variant exposed to the target. For RESUME, the call in step (3) must also use a new variant exposed to the target. The host implementation (7c) in Figure 16 is never used since RESUME is never invoked in the host.

We expose INTERRUPT and RESUME to the target by adding them to the target language as new constructs.

$$e ::= \textbf{interrupt}\ e_1\ e_2\ e_3 \mid \textbf{resume}\ e \tag{8}$$

We implement this functionality by augmenting the CPS evaluator with new clauses for $\mathcal{E}$ (Figure 17), clause (6p) for **interrupt** and clause (6q) for **resume**. We discuss the implementation of these below. But we first address several other issues.

$$\begin{aligned}
\mathcal{I}\ f\ l &= \langle(\lambda x.(\mathbf{interrupt}\ f\ x\ l)), \rho_0[f \mapsto f][l \mapsto l]\rangle && \text{(6n)}\\
\mathcal{R} &= \langle(\lambda z.(\mathbf{resume}\ z)), \rho_0\rangle && \text{(6o)}\\
\mathcal{E}\ k\ n\ l\ \rho\ (\mathbf{interrupt}\ e_1\ e_2\ e_3) &= \mathcal{E}\ (\lambda n\ l\ v_1. && \text{(6p)}\\
&\quad (\mathcal{E}\ (\lambda n\ l\ v_2. \\
&\qquad (\mathcal{E}\ (\lambda n\ l\ v_3. \\
&\qquad\quad \mathbf{if}\ l = \infty \\
&\qquad\quad \mathbf{then}\ (\mathcal{A}\ k\ 0\ v_3\ v_1\ v_2) \\
&\qquad\quad \mathbf{else\ let}\ [\![k, f]\!] = (\mathcal{A}\ k\ 0\ l\ v_1\ v_2) \\
&\qquad\qquad \mathbf{in}\ [\![k, (\mathcal{I}\ f\ (v_3 - l))]\!]) \\
&\qquad\ n\ l\ \rho\ e_3)) \\
&\quad\ n\ l\ \rho\ e_2)) \\
&\quad (n+1)\ l\ \rho\ e_1 \\
\mathcal{E}\ k\ n\ l\ \rho\ (\mathbf{resume}\ e) &= \mathcal{E}\ (\lambda n\ l\ [\![k', f]\!].(\mathcal{A}\ k'\ 0\ l\ f\ \bot))\ (n+1)\ l\ \rho\ e && \text{(6q)}
\end{aligned}$$

Figure 17. Additions to the CPS evaluator for CHECKPOINTVLAD to expose the general-purpose interruption and resumption interface to the target.

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

$$\begin{array}{lll}
\textbf{base case:} & (y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y} & \text{(step 0)}\\
\textbf{inductive case:} & l = \text{PRIMOPS}\ f\ x & \text{(step 1)}\\
 & z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor & \text{(step 2)}\\
 & (y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ \mathcal{R}\ z\ \grave{y} & \text{(step 3)}\\
 & (z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\mathcal{I}\ f\ \lfloor \frac{l}{2} \rfloor)\ x\ \grave{z} & \text{(step 4)}
\end{array}$$

Figure 18. Binary bisection checkpointing in the CPS evaluator. This is a proper implementation of the algorithm in Figure 13 where the host closure ($\lambda z.$RESUME z) in step (3) is replaced with the target closure $\mathcal{R}$ and the host closure ($\lambda x.$INTERRUPT $f\ x\ \lfloor l/2 \rfloor$) in step (4) is replaced with the target closure $(\mathcal{I}\ f\ \lfloor l/2 \rfloor)$.

With appropriate implementations of **interrupt** and **resume** expressions in the target language, one can create target closures for the expressions ($\lambda z.$RESUME z) and ($\lambda x.$INTERRUPT $f\ x\ \lfloor l/2 \rfloor$), and use these to formulate a proper implementation of $\overset{\checkmark}{\mathcal{J}}$ in the host. We formulate a target closure to correspond to ($\lambda z.$RESUME z) and denote this as $\mathcal{R}$. The definition is given in (6o) in Figure 17. Note that since ($\lambda z.$RESUME z) does not contain any free variables, the closure created by $\mathcal{R}$ is constructed from the empty environment ρ_0. Thus there is a single constant $\mathcal{R}$. We similarly formulate a target closure to correspond to ($\lambda x.$INTERRUPT $f\ x\ l$) and denote this as $\mathcal{I}$. The definition is given in (6n) in Figure 17. Here, however, ($\lambda x.$INTERRUPT $f\ x\ l$) contains two free variables: f and l. Thus the closure created by $\mathcal{I}$ contains a non-empty environment with values for these two variables. To provide these values, $\mathcal{I}$ is formulated as a function that takes these values as arguments.

With $(\mathcal{I}\ f\ l)$ and $\mathcal{R}$, it is now possible to reformulate the definition of $\overset{\checkmark}{\mathcal{J}}$ in the host from Figure 13, replacing the host closure ($\lambda z.$RESUME z) in step (3) with the target closure $\mathcal{R}$ and the host closure $(\lambda x.\textsc{interrupt}\, f\ x\ \lfloor l/2 \rfloor)$ in step (4) with the target closure $(\mathcal{I}\ f\ \lfloor l/2 \rfloor)$. This new, proper, definition of $\overset{\checkmark}{\mathcal{J}}$ in the host is given in Figure 18.

In this proper implementation of $\overset{\checkmark}{\mathcal{J}}$ in the host, the **interrupt** and **resume** operations need to be able to nest, even without nesting of calls to $\overset{\checkmark}{\mathcal{J}}$ in the target. The recursive calls to $\overset{\checkmark}{\mathcal{J}}$ in the inductive case of Figure 18 imply that it must be possible to interrupt a resumed capsule. This happens when passing $\mathcal{R}$ for f in step (3) and then passing $(\mathcal{I}\, f\ \ldots)$ for f in step (4), i.e. the left branch of a right branch in the checkpoint tree. The resulting function $f = (\mathcal{I}\ \mathcal{R}\ \ldots)$ will interrupt when applied to some capsule. It also happens when passing $(\mathcal{I}\, f\ \ldots)$ for f twice in succession in step (4), i.e. the left branch of a left branch in the checkpoint tree. The resulting function $f = (\mathcal{I}\ (\mathcal{I}\, f\ \ldots)\ \ldots)$ will interrupt and the capsule produced will interrupt when resumed.

Consider all the ways that evaluations of **interrupt** and **resume** expressions can nest. User code will never contain **interrupt** and **resume** expressions; they are created only by invocations of $\mathcal{I}$ and $\mathcal{R}$. $\mathcal{R}$ is only invoked by step (3) of $\overset{\checkmark}{\mathcal{J}}$ in Figure 18. $\mathcal{I}$ is invoked two ways: step (4) of $\overset{\checkmark}{\mathcal{J}}$ in Figure 18 and a way that we have not yet encountered, evaluation of nested **interrupt** expressions in the **else** branch of clause (6p) in Figure 17.

Consider all the ways that evaluations of $\mathcal{I}$ and $\mathcal{R}$ can be invoked in $\overset{\checkmark}{\mathcal{J}}$ in Figure 18. $\overset{\checkmark}{\mathcal{J}}$ is invoked with some user code for f, i.e. code that does not contain **interrupt** and **resume** expressions. The inductive cases for $\overset{\checkmark}{\mathcal{J}}$ create a binary checkpoint tree of invocations. The leaf nodes of this binary checkpoint tree correspond to the base case in step (0) where the host $\overleftarrow{\mathcal{J}}$ is invoked. At internal nodes, the host INTERRUPT is invoked in step (2). The target closure values that can be passed to the host $\overleftarrow{\mathcal{J}}$ and INTERRUPT are constructed from f, $\mathcal{I}$, and $\mathcal{R}$ in steps (3) and (4). What is the space of all possible constructed target closures? The constructed target closures invoked along the left spine of the binary checkpoint tree look like the following

$$(\mathcal{I}\ (\mathcal{I}\ \cdots\ (\mathcal{I}\ (\mathcal{I}\, f\ l_0)\ l_1)\ \cdots\ l_{i-1})\ l_i) \tag{9}$$

with zero or more nested calls to $\mathcal{I}$. In this case $l_i < l_{i-1} < \cdots < l_1 < l_0$, because the recursive calls to $\overset{\checkmark}{\mathcal{J}}$ in step (4) of Figure 18 always reduce l. The constructed target closures invoked in any other node in the binary checkpoint tree look like the following

$$(\mathcal{I}\ (\mathcal{I}\ \cdots\ (\mathcal{I}\ (\mathcal{I}\ \mathcal{R}\ l_0)\ l_1)\ \cdots\ l_{i-1})\ l_i) \tag{10}$$

with zero or more nested calls to $\mathcal{I}$. In this case, again, $l_i < l_{i-1} < \cdots < l_1 < l_0$, for the same reason. These are the possible target closures f passed to $\overleftarrow{\mathcal{J}}$ in step (0) or INTERRUPT in step (2) of $\overset{\checkmark}{\mathcal{J}}$ in Figure 18. (We assume that the call to PRIMOPS in step (1) is hoisted out of the recursion.)

A string of calls to $\mathcal{I}$ as in (9) will result in a nested closure structure whose invocation will lead to nested invocations of **interrupt** expressions.

$$\begin{aligned}
&\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\quad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\quad\rho_0[f\mapsto\langle\ldots\\
&\qquad\qquad\quad(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\qquad\quad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\qquad\qquad\quad\rho_0[f\mapsto f][l\mapsto l_0]\rangle][l\mapsto l_1]\ldots\rangle][l\mapsto l_{i-1}]\rangle][l\mapsto l_i]\rangle
\end{aligned}\tag{11}$$

A string of calls to $\mathcal{I}$ as in (10) will also result in a nested closure structure whose invocation will lead to nested invocations of **interrupt** expressions.

$$\begin{aligned}
&\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\quad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\quad\rho_0[f\mapsto\langle\ldots\\
&\qquad\qquad\quad(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\qquad\quad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\qquad\qquad\quad\rho_0[f\mapsto\langle(\lambda z.(\textbf{resume}\ z)),\rho_0\rangle]\\
&\qquad\qquad\qquad\qquad[l\mapsto l_0]\rangle][l\mapsto l_1]\ldots\rangle][l\mapsto l_{i-1}]\rangle][l\mapsto l_i]\rangle
\end{aligned}\tag{12}$$

In both of these, $l_i < l_{i-1} < \cdots < l_1 < l_0$, so the outermost **interrupt** expression will interrupt first. Since the CPS evaluator only maintains a single step limit, l_i will be that step limit during the execution of the innermost content of these nested closures, namely f in (11) and $\langle(\lambda z.(\textbf{resume}\ z)),\rho_0\rangle$ in (12). None of the other intervening **interrupt** expressions will enforce their step limits during this execution. Thus we need to arrange for the capsule created when the step limit l_i is reached during the execution of f or $\langle(\lambda z.(\textbf{resume}\ z)),\rho_0\rangle$ to itself interrupt with the remaining step limits $l_{i-1},\ldots,l_1,l_0$. This is done by rewrapping the closure in a capsule with **interrupt** expressions. The interruption of f or $\langle(\lambda z.(\textbf{resume}\ z)),\rho_0\rangle$ will produce a capsule that looks like the following

$$[\![k,f]\!]\tag{13}$$

where the closure f contains only user code, i.e. no **interrupt** or **resume** expressions. The f in (13) is wrapped with calls to $\mathcal{I}$ to reintroduce the step limits $l_{i-1},\ldots,l_1,l_0$.

$$[\![k,(\mathcal{I}\ \cdots\ (\mathcal{I}\ (\mathcal{I}\, f\ l_0)\ l_1)\ \cdots\ l_{i-1})]\!]\tag{14}$$

This will yield a capsule that looks like the following

$$\begin{aligned}
&[\![k,\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\rho_0[f\mapsto\langle\ldots\\
&\qquad\qquad(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\qquad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
&\qquad\qquad\qquad\rho_0[f\mapsto f][l\mapsto l_0]\rangle][l\mapsto l_1]\ldots\rangle][l\mapsto l_{i-1}]\rangle]\!]
\end{aligned}\tag{15}$$

which will interrupt upon resumption. Each such interruption will peel off one **interrupt** expression. Note that since the closure f in a capsule (13) contains only user code, it will not contain a **resume** expression. Further, since the wrapping process (15) only introduces **interrupt** expressions via calls to $\mathcal{I}$ (14), and never introduces **resume** expressions, the closures in capsules, whether wrapped or not, will never contain **resume** expressions.

When there is no contextual step limit, i.e. when $l = \infty$, the **interrupt** expression must introduce v_3, the step limit specified as the argument to the **interrupt** expression, as the step limit. This is handled by the **then** branch of clause (6p) in Figure 17. When there is a contextual step limit, i.e. when $l \neq \infty$, the **interrupt** expression must wrap the returned capsule. This wrapping is handled by the **else** branch of clause (6p) in Figure 17. Since capsule resumption restarts the step count at zero, the wrapping that handles nested step limits is relativized to this restart by the $v_3 - l$ in the **else** branch in clause (6p).

Capsule resumption happens in one place, the call to $\mathcal{A}$ in clause (6q) in Figure 17 for a **resume** expression. Except for the contextual step limit l, this is the same as the call to $\mathcal{A}$ in the implementation of RESUME in (7c) in Figure 16. Said resumption is performed by applying the capsule closure f, a target closure, to $\perp$, since the lambda expression in the capsule closure ignores its argument. This call to $\mathcal{A}$ is passed the capsule continuation k' as its continuation. Unlike the implementation of RESUME in (7c), the step limit l is that which is in effect for the execution of the **resume** expression. This is to allow capsule resumption to itself interrupt. Because capsules are resumed with a step count of zero and the step limit at the time of resumption, the step count and limit at the time of the interruption need not be saved in the capsule.

As a result of this, all **interrupt** expressions will appear in one of two places. The first is a preamble (11) or (12) wrapped around either a user function f by (9) or a **resume** expression in $\mathcal{R}$ by (10), respectively. This will always be invoked either by $\overleftarrow{\mathcal{J}}$ in the base case, step (0), or by INTERRUPT in step (2), of Figure 18. The second is a preamble (15) wrapped around the closure of a capsule by the **else** branch in clause (6p) of Figure 17, i.e. (14). This will always be invoked during capsule resumption, i.e. clause (6q) of Figure 17. We assume that the step limits are such that an interruption never occurs during either of these preambles. This is enforced by ensuring that the termination criterion that triggers the base case, step (0), of Figure 18 is sufficiently long so that the calls to $\mathcal{A}$ in $\overleftarrow{\mathcal{J}}$ in step (0) and INTERRUPT in step (2) won't interrupt before completion of the preamble.

There is one further requirement to allow the CPS evaluator to support divide-and-conquer checkpointing. The base case use of $\overleftarrow{\mathcal{J}}$ in step (0) of Figure 18 needs to be able to produce cotangents $\grave{z}$ of capsules z in step (3) and consume them in step (4). A capsule $[\![k, f]\!]$ is the saved state of the evaluator. The value f is a target closure $\langle(\lambda x.e), \rho\rangle$ which contains an environment with saved state. This state is visible to $\overleftarrow{\mathcal{J}}$. But the continuation k is a host continuation, which is opaque. Any evaluator variables that it closes over are not visible to $\overleftarrow{\mathcal{J}}$. Thus the implementation of host continuations in the CPS evaluator must employ a mechanism to expose them. When we replace the CPS evaluator with a direct-style evaluator applied to CPS-converted target code, in Sections 3.11 and 3.12 , this will no-longer be necessary since continuations will be represented as target closures which are visible to $\overleftarrow{\mathcal{J}}$.

3.9. Augmenting the CPS evaluator to support divide-and-conquer checkpointing

We can now add the $\overset{\checkmark}{\mathcal{J}}$ operator to the target language as a new construct.

$$e ::= \overset{\checkmark}{\mathcal{J}}\ e_1\ e_2\ e_3 \tag{16}$$

$$
\begin{aligned}
\mathcal{E}\; k\; n\; l\; \rho\; (\overset{\checkmark}{\mathcal{J}}\; e_1\; e_2\; e_3) = \mathcal{E}\; (\lambda n\; l\; v_1.&\\
&(\mathcal{E}\; (\lambda n\; l\; v_2.\\
&\quad (\mathcal{E}\; (\lambda n\; l\; v_3.\\
&\qquad\quad (k\; n\; l\; (\overset{\checkmark}{\mathcal{J}}\; v_1\; v_2\; v_3)))\\
&\qquad n\; l\; \rho\; e_3))\\
&\quad n\; l\; \rho\; e_2))\\
&(n+1)\; l\; \rho\; e_1
\end{aligned}
\tag{6r}
$$

Figure 19. Addition to the CPS evaluator for CHECKPOINTVLAD to support divide-and-conquer checkpointing.

We implement this functionality by augmenting the CPS evaluator with a new clause (6r) for $\mathcal{E}$ (Figure 19).

With this addition, target programs can perform divide-and-conquer checkpointing simply by calling $\overset{\checkmark}{\mathcal{J}}$ instead of $\overleftarrow{\mathcal{J}}$. Note that it is not possible to add the $\overset{\checkmark}{\mathcal{J}}$ operator to the direct-style evaluator because the implementation of binary bisection checkpointing is built on the general-purpose interruption and resumption interface which is, in turn, built on the CPS evaluator. We remove this limitation below in Section 3.12. Also note that since the implementation of binary bisection checkpointing is built on the general-purpose interruption and resumption interface which is, in turn, built on an evaluator, it is only available for programs that are evaluated, i.e. for programs in the target, but not for programs in the host. We remove this limitation below as well, in Section 3.13.

3.10. Some intuition

The algorithm in Figure 18 corresponds to Figure 2(b). The start of the computation of f in Figure 18 corresponds to u in Figure 2(b). The computation state at u is x in Figure 18. Collectively, the combination of f and x in Figure 18 comprises a snapshot, the gold line in Figure 2(b). The end of the computation of f in Figure 18 corresponds to v in Figure 2(b). The computation state at v is y in Figure 18. Step (1) computes $\lfloor l/2 \rfloor$ which corresponds to the split point p in Figure 2(b). Step (2) corresponds to the green line in Figure 2(b), i.e. running the primal without taping from the snapshot f and x at u until the split point p which is $\lfloor l/2 \rfloor$. The capsule z in Figure 18 corresponds to the computation state at p in Figure 2(b). Brown and pink lines in Figure 2 denote capsules. If step (3) would incur the base case, step (0), in the recursive call, it would correspond to the right stage (pair of red and blue lines) in Figure 2(b). If step (4) would incur the base case, step (0), in the recursive call, it would correspond to the left stage (pair of red and blue lines) in Figure 2(b). Note that f and x is used both in steps (2) and (4). Referring to this as a snapshot is meant to convey that the information must be saved across the execution of step (3). And it must be possible to apply f to x twice, once in step (2) and once in step (4). In some implementations, such a snapshot involves saving mutable state that must be restored. In our formulation in a functional framework (Section 7.2), we need not explicitly save and restore state; we simply apply a function twice. Nonetheless, the storage required for the snapshot is implicit in the extended lifetime of the values f and x which extends from the entry into $\overset{\checkmark}{\mathcal{J}}$, over step (3), until step (4).

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x \tag{17a}$$
$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner) \tag{17b}$$
$$\ulcorner (e_1\ e_2)|k \urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner \tag{17c}$$
$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner \tag{17d}$$

Figure 20. CPS conversion for the untyped lambda calculus.

Note that recursive calls to $\overset{\checkmark}{\mathcal{J}}$ in step (4) extend the lifetime of a snapshot. These are denoted as the black tick marks on the left of the gold and pink lines. In the treeverse algorithm from [9, Figures 2 and 3], the lifetime of one snapshot ends at a tick mark by a call to **retrieve** in one recursive call to **treeverse** in the **while** loop of the parent and the lifetime of a new snapshot begins by a call to **snapshot** in the next recursive call to **treeverse** in the **while** loop of the parent. But since the state retrieved and then immediately saved again as a new snapshot is the same, these adjacent snapshot execution intervals can conceptually be merged.

Also note that recursive calls to $\overset{\checkmark}{\mathcal{J}}$ in step (3) pass $\mathcal{R}$ and a capsule z as the f and x of the recursive call. Thus capsules from one level of the recursion become snapshots at the next level, for all but the base case step (0). Pink lines in Figure 2 denote values that are capsules at one level but snapshots at lower levels. Some, but not all, capsules are snapshots. Some, but not all, snapshots are capsules. Gold lines in Figure 2 denote snapshots that are not capsules. Brown lines in Figure 2 denote capsules that are not snapshots. Pink lines in Figure 2 denote values that are both snapshots and capsules.

It is now easy to see that the recursive call tree of the algorithm in Figure 18 is isomorphic to a binary checkpoint tree. The binary checkpoint tree in Figure 3 corresponds to the call tree in Figure 21 produced by the algorithm in Figure 18. This depicts just one level of the recursion. If one unrolls this call tree, one obtains a binary checkpoint tree.

3.11. CPS conversion

So far, we have formulated divide-and-conquer checkpointing via a CPS evaluator. This can be – and has been – used to construct an interpreter. A compiler can be – and has been – constructed by generating target code in CPS that is instrumented with step counting, step limits, and limit checks that lead to interrupts. Code in direct style can be automatically converted to CPS using a program transformation known in the programming language community as *CPS conversion*. Many existing compilers, such as SML/NJ for SML, perform CPS conversion as part of the compilation process [4].

We illustrate CPS conversion for the untyped lambda calculus (Figure 20).

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \tag{18}$$

The notation $\ulcorner e|k \urcorner$ denotes the transformation of the expression e to CPS so that it calls the continuation k with the result. There is a clause for $\ulcorner e|k \urcorner$ in Figure 20, (17a–c), for each construct in (18). Clause (17a) says that one converts a variable x by calling the continuation k with the value of that variable. Clause (17b) says that one converts a lambda expression

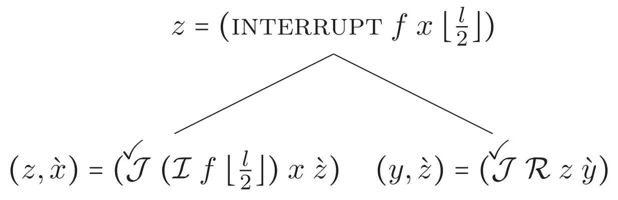
$z = (\text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor)$
$(z, \grave{x}) = (\overset{\checkmark}{\mathcal{J}}\ (\mathcal{I}\ f\ \lfloor \frac{l}{2} \rfloor)\ x\ \grave{z})$
$(y, \grave{z}) = (\overset{\checkmark}{\mathcal{J}}\ \mathcal{R}\ z\ \grave{y})$

Figure 21. The call tree produced by the algorithm in Figure 18 that yields the binary checkpoint tree in Figure 3.

$(\lambda x.e)$ by adding a continuation variable k' to the lambda binder, converting the body relative to that variable, and then calling the continuation k with that lambda expression. Clause (17c) says that one converts an application $(e_1\ e_2)$ by converting e_1 with a continuation that receives the value x_1 of e_1, then converts e_2 with a continuation that receives the value x_2 of e_2, and then calls x_1 with the continuation k and x_2. Clause (17d) says that the top level expression e_0 can be converted with the identity function as the continuation.

This technique can be extended to thread a step count n and a step limit l through the computation along with the continuation k, and to arrange for the step count to be incremented appropriately. Further, this technique can be applied to the entire target language (Figure 22). Clauses (20a–j) correspond one-to-one to the CHECKPOINTVLAD constructs in (1), (4), and (16). Since CPS conversion is only applied once at the beginning of compilation, to the user program, and the user program does not contain **interrupt** and **resume** expressions, since these only appear internally in the target closures created by $\mathcal{I}$ and $\mathcal{R}$, CPS conversion need not handle these constructs. Finally, $\langle\!\langle e \rangle\!\rangle_{k,n,l}$ denotes a limit check that interrupts and returns a capsule when the step count n reaches the step limit l. The implementation of this limit check is given in (20k). Each of the clauses (20a–j) is wrapped in a limit check.

3.12. Augmenting the direct-style evaluator to support CPS-converted code and divide-and-conquer checkpointing

The direct-style evaluator must be modified in several ways to support CPS-converted code and divide-and-conquer checkpointing (Figure 23). First, CPS conversion introduced lambda expressions with multiple arguments and their corresponding applications. Continuations have three arguments and converted lambda expressions have four. Thus we add several new constructs into the target language to replace the single argument lambda expressions and applications from (1).

$$e ::= \lambda_3 n\ l\ x.e \mid \lambda_4 k\ n\ l\ x.e \mid e_1\ e_2\ e_3\ e_4 \mid e_1\ e_2\ e_3\ e_4\ e_5 \qquad (19)$$

Second, we need to modify $\mathcal{E}$ to support these new constructs. We replace clause (2a) with clauses (21a, 21b) to update $\mathcal{A}$ and clauses (2d, 2e) with clauses (21c–f) to update $\mathcal{E}$. Third, we need to add support for **interrupt** and **resume** expressions, as is done with clauses (21g, 21h). These are direct-style variants of clauses (6p, 6q) from the CPS evaluator and are needed to add support for the general-purpose interruption and resumption interface to the direct-style evaluator when evaluating CPS code. Note that the calls to $\mathcal{A}$ from (6p, 6q) are modified to use the converted form $\mathcal{A}_4$ of $\mathcal{A}$ (21b) in (21g, 21h). Similarly, the calls

$$\ulcorner c|k,n,l\urcorner \rightsquigarrow \langle\!\langle k\ (n+1)\ l\ c\rangle\!\rangle_{k,n,l} \tag{20a}$$

$$\ulcorner x|k,n,l\urcorner \rightsquigarrow \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l} \tag{20b}$$

$$\ulcorner(\lambda x.e)|k,n,l\urcorner \rightsquigarrow \langle\!\langle k\ (n+1)\ l\ (\lambda_4 k\ n\ l\ x.\ulcorner e|k,n,l\urcorner)\rangle\!\rangle_{k,n,l} \tag{20c}$$

$$\begin{aligned}\ulcorner(e_1\ e_2)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\ &\ulcorner e_2|(\lambda_3 n\ l\ x_2.\\ &\qquad (x_1\ k\ n\ l\ x_2)),\\ &\quad n,l\urcorner),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20d}$$

$$\begin{aligned}\ulcorner(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\ &(\textbf{if}\ x_1\\ &\ \ \textbf{then}\ \ulcorner e_2|k,n,l\urcorner\\ &\ \ \textbf{else}\ \ulcorner e_3|k,n,l\urcorner)),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20e}$$

$$\begin{aligned}\ulcorner(\diamond e)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e|(\lambda_3 n\ l\ x.\\ &(k\ n\ l\ (\diamond x))),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20f}$$

$$\begin{aligned}\ulcorner(e_1 \bullet e_2)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\ &\ulcorner e_2|(\lambda_3 n\ l\ x_2.\\ &\qquad (k\ n\ l\ (x_1 \bullet x_2))),\\ &\quad n,l\urcorner),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20g}$$

$$\begin{aligned}\ulcorner(\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\ &\ulcorner e_2|(\lambda_3 n\ l\ x_2.\\ &\qquad \ulcorner e_3|(\lambda_3 n\ l\ x_3.\\ &\qquad\qquad (k\ n\ l\ (\overrightarrow{\mathcal{J}}\ x_1\ x_2\ x_3))),\\ &\qquad\quad n,l\urcorner),\\ &\quad n,l\urcorner),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20h}$$

$$\begin{aligned}\ulcorner(\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\ &\ulcorner e_2|(\lambda_3 n\ l\ x_2.\\ &\qquad \ulcorner e_3|(\lambda_3 n\ l\ x_3.\\ &\qquad\qquad (k\ n\ l\ (\overleftarrow{\mathcal{J}}\ x_1\ x_2\ x_3))),\\ &\qquad\quad n,l\urcorner),\\ &\quad n,l\urcorner),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20i}$$

$$\begin{aligned}\ulcorner(\overset{\checkmark}{\mathcal{J}}\ e_1\ e_2\ e_3)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\ &\ulcorner e_2|(\lambda_3 n\ l\ x_2.\\ &\qquad \ulcorner e_3|(\lambda_3 n\ l\ x_3.\\ &\qquad\qquad (k\ n\ l\ (\overset{\checkmark}{\mathcal{J}}\ x_1\ x_2\ x_3))),\\ &\qquad\quad n,l\urcorner),\\ &\quad n,l\urcorner),\\ &(n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned} \tag{20j}$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if}\ n=l\ \textbf{then}\ [\![k,\lambda_4 k\ n\ l\ \_.e]\!]\ \textbf{else}\ e \tag{20k}$$

Figure 22. CPS conversion for the CHECKPOINTVLAD language that threads step counts and limits.

$$\mathcal{A}_3\ \langle(\lambda_3 n\ l\ x.e),\rho\rangle\ n'\ l'\ v = \mathcal{E}\ \rho[n\mapsto n'][l\mapsto l'][x\mapsto v]\ e \tag{21a}$$

$$\mathcal{A}_4\ \langle(\lambda_4 k\ n\ l\ x.e),\rho\rangle\ k'\ n'\ l'\ v = \mathcal{E}\ \rho[k\mapsto k'][n\mapsto n'][l\mapsto l'][x\mapsto v]\ e \tag{21b}$$

$$\mathcal{E}\ \rho\ (\lambda_3 n\ l\ x.e) = \langle(\lambda_3 n\ l\ x.e),\rho\rangle \tag{21c}$$

$$\mathcal{E}\ \rho\ (\lambda_4 k\ n\ l\ x.e) = \langle(\lambda_4 k\ n\ l\ x.e),\rho\rangle \tag{21d}$$

$$\mathcal{E}\ \rho\ (e_1\ e_2\ e_3\ e_4) = \mathcal{A}_3\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)\ (\mathcal{E}\ \rho\ e_4) \tag{21e}$$

$$\mathcal{E}\ \rho\ (e_1\ e_2\ e_3\ e_4\ e_5) = \mathcal{A}_4\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)\ (\mathcal{E}\ \rho\ e_4)\ (\mathcal{E}\ \rho\ e_5) \tag{21f}$$

$$\mathcal{E}\rho\ (\textbf{interrupt}\ e_1\ e_2\ e_3) = \begin{array}{l}\textbf{let}\ v_1 = (\mathcal{E}\ \rho\ e_1)\\ \quad\ \ v_2 = (\mathcal{E}\ \rho\ e_2)\\ \quad\ \ v_3 = (\mathcal{E}\ \rho\ e_3)\\ \quad\ \ k = \rho\ \text{‘k’}\\ \quad\ \ l = \rho\ \text{‘l’}\\ \textbf{in if}\ l = \infty\\ \quad \textbf{then}\ (\mathcal{A}_4\ v_1\ k\ 0\ v_3\ v_2)\\ \quad \textbf{else let}\ [\![k,f]\!] = (\mathcal{A}_4\ v_1\ k\ 0\ l\ v_2)\\ \qquad\quad \textbf{in}\ [\![k,(\mathcal{I}\ f\ (v_3 - l))]\!]\end{array} \tag{21g}$$

$$\mathcal{E}\ \rho\ (\textbf{resume}\ e) = \begin{array}{l}\textbf{let}\ [\![k',f]\!] = (\mathcal{E}\ \rho\ e)\\ \quad\ \ l = \rho\ \text{‘l’}\\ \textbf{in}\ (\mathcal{A}_4\ f\ k'\ 0\ l\ \bot)\end{array} \tag{21h}$$

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \begin{array}{l}\textbf{let}\ (v_4 \triangleright \acute{v}_5) = (\mathcal{A}_4\ v_1\ \langle(\lambda_3 n\ l\ v.v),\rho_0\rangle\ 0\ \infty\ (v_2 \triangleright \acute{v}_3))\\ \textbf{in}\ (v_4,\acute{v}_5)\end{array} \tag{21i}$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \begin{array}{l}\textbf{let}\ (v_4 \triangleleft \grave{v}_5) = ((\mathcal{A}_4\ v_1\ \langle(\lambda_3 n\ l\ v.v),\rho_0\rangle\ 0\ \infty\ v_2) \triangleleft \grave{v}_3)\\ \textbf{in}\ (v_4,\grave{v}_5)\end{array} \tag{21j}$$

$$\textsc{primops}\ f\ x = \mathcal{A}_4\ f\ \langle(\lambda_3 n\ l\ v.n),\rho_0\rangle\ 0\ \infty\ x \tag{21k}$$

$$\textsc{interrupt}\ f\ l\ n = \mathcal{A}_4\ f\ \langle(\lambda_3 n\ l\ v.v),\rho_0\rangle\ 0\ l\ x \tag{21l}$$

$$\mathcal{I}\ f\ l = \langle(\lambda_4 k\ n\ l\ x.(\textbf{interrupt}\ f\ x\ l)),\rho_0[f\mapsto f][l\mapsto l]\rangle \tag{21m}$$

$$\mathcal{R} = \langle(\lambda_4 k\ n\ l\ z.(\textbf{resume}\ z)),\rho_0\rangle \tag{21n}$$

Figure 23. Extensions to the direct-style evaluator and the implementation of the general-purpose interruption and resumption interface to support divide-and-conquer checkpointing on target code that has been converted to CPS.

to continuations from (6p, 6q) are modified to use the continuation form $\mathcal{A}_3$ of $\mathcal{A}$ (21a) in (21g, 21h). Fourth, the calls to $\mathcal{A}_4$ must be modified in the host implementations of the AD operators $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$, as is done with (21i, 21j). Note that unlike the corresponding (2i, 2j), the calls to $\mathcal{A}_4$ here take target closures instead of host closures. Fifth, the general-purpose interruption and resumption interface, (7a, 7b, 6n, 6o), must be migrated from the CPS evaluator to the direct-style evaluator as (21k–n). In doing so, the calls to $\mathcal{A}_4$ in PRIMOPS and INTERRUPT are changed to use (21b), the host continuations are modified to be target continuations in (21k, 21l), and the lambda expressions in (21m, 21n) are CPS converted.

3.13. Compiling direct-style code to C

One can compile target CHECKPOINTVLAD code, after CPS conversion, to C (Figures 24 and 25). Modern implementations of C, like GCC, together with modern memory management technology, like the Boehm–Demers–Weiser garbage collector, allow the compilation process to be a straightforward mapping of each construct to a small

$$\mathcal{S}\;\pi\;() = \texttt{null\_constant} \tag{22a}$$

$$\mathcal{S}\;\pi\;\mathbf{true} = \texttt{true\_constant} \tag{22b}$$

$$\mathcal{S}\;\pi\;\mathbf{false} = \texttt{false\_constant} \tag{22c}$$

$$\mathcal{S}\;\pi\;(c_1, c_2) = \texttt{cons((}\mathcal{S}\;\pi\;c_1\texttt{), (}\mathcal{S}\;\pi\;c_1\texttt{))} \tag{22d}$$

$$\mathcal{S}\;\pi\;n = n \tag{22e}$$

$$\mathcal{S}\;\pi\;\texttt{`k'} = \texttt{continuation} \tag{22f}$$

$$\mathcal{S}\;\pi\;\texttt{`n'} = \texttt{count} \tag{22g}$$

$$\mathcal{S}\;\pi\;\texttt{`l'} = \texttt{limit} \tag{22h}$$

$$\mathcal{S}\;\pi\;\texttt{`x'} = \texttt{argument} \tag{22i}$$

$$\mathcal{S}\;\pi\;x = \texttt{as\_closure(target)->environment[}\pi\;x\texttt{]} \tag{22j}$$

$\mathcal{S}\;\pi\;(\lambda_3 n\;l\;x.e)$ = ({ (22k)

```
thing function(thing target,
               thing count,
               thing limit,
               thing argument) {
return ($\mathcal{S}\;(\phi\;e)\;e$);
}
thing lambda = (thing)GC_malloc(sizeof(struct {
  enum tag tag;
  struct {
    thing (*function)();
    unsigned n;
    thing environment[$|\phi\;e|$];
  }))
}
set_closure(lambda);
as_closure(lambda)->function = &function;
as_closure(lambda)->n = $|\phi\;e|$;
as_closure(lambda)->environment[0] = $\mathcal{S}\;\pi\;(\phi\;e)_0$
⋮
as_closure(lambda)->environment[$|\phi\;e|-1$] = $\mathcal{S}\;\pi\;(\phi\;e)_{|\phi\;e|-1}$
lambda;
})
```

Figure 24. Compiler for the CHECKPOINTVLAD language when in CPS. Part I.

fragment of C code. In particular, garbage collection, GC_malloc, eases the implementation of closures and statement expressions, ({...}), together with nested functions, ease the implementation of lambda expressions. Furthermore, the flow analysis, inlining, and tail-call merging performed by GCC generates reasonably efficient code. In Figures 24 and 25, $\mathcal{S}$ denotes such a mapping from CHECKPOINTVLAD expressions e to C code fragments. Instead of environments ρ, $\mathcal{S}$ takes π, a mapping from variables to indices in environment, the run-time environment data structure. Here, $\pi\;x$ denotes the index of x, π_i denotes the variable for index i, $\phi\;e$ denotes a mapping for the free variables in e, and $\mathcal{N}$ denotes a mapping from a CHECKPOINTVLAD operator to the name of the C function that implements that operator. This, together with a library containing the typedef for thing, the enum for tag, definitions for null_constant, true_constant, false_constant, cons, as_closure, set_closure, continuation_apply, converted_apply, is_false, and all of the functions named by $\mathcal{N}$ (essentially a translation of R6RS-AD, the general-purpose interruption and resumption interface from Figure 23, and the implementation of binary bisection checkpointing from Figure 18 into C), allows arbitrary CHECKPOINTVLAD code

```
$\mathcal{S}\ \pi\ (\lambda_4 k\ n\ l\ x.e)$ = ({                                                    (22l)
                           thing function(thing target,
                                          thing continuation,
                                          thing count,
                                          thing limit,
                                          thing argument) {
                           return ($\mathcal{S}\ (\phi\ e)\ e$);
                           }
                           thing lambda = (thing)GC_malloc(sizeof(struct {
                             enum tag tag;
                             struct {
                               thing (*function)();
                               unsigned n;
                               thing environment[$|\phi\ e|$];
                             }))
                           }
                           set_closure(lambda);
                           as_closure(lambda)->function = &function;
                           as_closure(lambda)->n = $|\phi\ e|$;
                           as_closure(lambda)->environment[0] = $\mathcal{S}\ \pi\ (\phi\ e)_0$
                           $\vdots$
                           as_closure(lambda)->environment[$|\phi\ e| - 1$] = $\mathcal{S}\ \pi\ (\phi\ e)_{|\phi\ e|-1}$
                           lambda;
                         })
$\mathcal{S}\ \pi\ (e_1\ e_2\ e_3\ e_4)$ = continuation_apply(($\mathcal{S}\ \pi\ e_1$),                       (22m)
                                            ($\mathcal{S}\ \pi\ e_2$),
                                            ($\mathcal{S}\ \pi\ e_3$),
                                            ($\mathcal{S}\ \pi\ e_4$))
$\mathcal{S}\ \pi\ (e_1\ e_2\ e_3\ e_4\ e_5)$ = converted_apply(($\mathcal{S}\ \pi\ e_1$),                     (22n)
                                          ($\mathcal{S}\ \pi\ e_2$),
                                          ($\mathcal{S}\ \pi\ e_3$),
                                          ($\mathcal{S}\ \pi\ e_4$),
                                          ($\mathcal{S}\ \pi\ e_5$))
$\mathcal{S}\ \pi\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3)$ = (!is_false(($\mathcal{S}\ \pi\ e_1$))?($\mathcal{S}\ \pi\ e_2$):($\mathcal{S}\ \pi\ e_3$))      (22o)
$\mathcal{S}\ \pi\ (\diamond e)$ = ($\mathcal{N}\ \diamond$)(($\mathcal{S}\ \pi\ e$))                                      (22p)
$\mathcal{S}\ \pi\ (e_1 \bullet e_2)$ = ($\mathcal{N}\ \bullet$)(($\mathcal{S}\ \pi\ e_1$), ($\mathcal{S}\ \pi\ e_2$))                    (22q)
$\mathcal{S}\ \pi\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3)$ = ($\mathcal{N}\ \overrightarrow{\mathcal{J}}$)(($\mathcal{S}\ \pi\ e_1$), ($\mathcal{S}\ \pi\ e_2$), ($\mathcal{S}\ \pi\ e_3$))    (22r)
$\mathcal{S}\ \pi\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3)$ = ($\mathcal{N}\ \overleftarrow{\mathcal{J}}$)(($\mathcal{S}\ \pi\ e_1$), ($\mathcal{S}\ \pi\ e_2$), ($\mathcal{S}\ \pi\ e_3$))    (22s)
$\mathcal{S}\ \pi\ (\overset{\checkmark}{\mathcal{J}}\ e_1\ e_2\ e_3)$ = ($\mathcal{N}\ \overset{\checkmark}{\mathcal{J}}$)(($\mathcal{S}\ \pi\ e_1$), ($\mathcal{S}\ \pi\ e_2$), ($\mathcal{S}\ \pi\ e_3$))    (22t)
```

Figure 25. Compiler for the CHECKPOINTVLAD language when in CPS. Part II.

to be compiled to machine code, via C, with complete support for AD, including forward mode, reverse mode, and binary bisection checkpointing.

3.14. Implementations

We have written three complete implementations of CHECKPOINTVLAD. All three accept exactly the same source language in its entirety and are able to run the example discussed in Section 6. The first implementation is an interpreter based on the CPS evaluator (Figures 14, 15, 17, and 19), where the evaluator, the operator overloading implementation of AD, the general-purpose interruption and resumption mechanism (Figure 16), and the binary bisection checkpointing driver (Figure 18) are implemented in SCHEME.

The second implementation is a hybrid compiler/interpreter that translates the checkpointVLAD source program into CPS using CPS conversion (Figure 22) and then interprets this with an interpreter based on the direct-style evaluator (Figures 9, 10, and 23), where the compiler, the evaluator, the operator overloading implementation of AD, the general-purpose interruption and resumption mechanism (Figure 23), and the binary bisection checkpointing driver (Figure 18) are implemented in Scheme. The third implementation is a compiler that translates the checkpointVLAD source program into CPS using CPS conversion (Figure 22) and then compiles this to machine code via c using GCC, where the compiler (Figures 24 and 25) is implemented in Scheme, the evaluator is the underlying hardware, and the operator overloading implementation of AD, the general-purpose interruption and resumption mechanism (Figure 23), and the binary bisection checkpointing driver (Figure 13) are implemented in c. The first implementation was used to generate the results reported in [20] and presented at AD (2016). The techniques of Figures 20 and 22 were presented at AD (2016). The third implementation was used to generate the results reported here.

4. Complexity

The internal nodes of a binary checkpoint tree correspond to invocations of interrupt in step (2). The right branches of each node correspond to step (3). The left branches of each node correspond to step (4). The leaf nodes correspond to invocations of $\overleftarrow{\mathcal{J}}$ in the base case, step (0). Each leaf node corresponds to a stage, the red, blue, and violet lines in Figure 2(g). The checkpoint tree is traversed in depth-first right-to-left preorder. In our implementation, we terminate the recursion when the step limit l is below a fixed constant. Consider a general primal computation f that uses maximal live storage w and that runs for t steps. This results in the following space and time complexities for reverse mode without checkpointing, including our implementation of $\overleftarrow{\mathcal{J}}$, and for binary bisection checkpointing, including our implementation of $\overset{\checkmark}{\mathcal{J}}$.

	Complexity									
	space					time				
Computation	primal	snapshots	tape	total	overhead	recomputation	forward sweep	reverse sweep	total	overhead
without checkpointing	$O(w)$		$O(t)$	$O(w+t)$	$O\left(\frac{w+t}{w}\right)$		$O(t)$	$O(t)$	$O(t)$	$O(1)$
binary bisection checkpointing	$O(w)$	$O(w\log t)$	$O(1)$	$O(w\log t)$	$O(\log t)$	$O(t\log t)$	$O(t)$	$O(t)$	$O(t\log t)$	$O(\log t)$

If we assume that $O(t) \geq O(w)$, i.e. that the computation uses all storage, the total space requirement without checkpointing becomes $O(t)$ and the overhead becomes $O(t)$.

5. Extensions to support treeverse and binomial checkpointing

The general-purpose interruption and resumption interface allows implementation of $\overset{\checkmark}{\mathcal{J}}$ using the treeverse algorithm from [9, Figure 4] as shown in Figure 26. This supports the

```
TREEVERSE $f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \beta\ \sigma\ \phi$ = if $\sigma > \beta$
                                        then let $z$ = INTERRUPT $f\ x\ (\sigma - \beta)$
                                            in FIRST $\mathcal{R}\ z\ \grave{y}\ \alpha\ (\delta - 1)\ \tau\ \beta\ \sigma\ \phi$
                                        else FIRST $f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \beta\ \sigma\ \phi$
FIRST $f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \beta\ \sigma\ \phi$ = if $\phi - \sigma > \alpha \wedge \delta \neq 0 \wedge \tau \neq 0$
                                        then let $\kappa$ = MID $\delta\ \tau\ \sigma\ \phi$
                                                 $(y, \grave{z})$ = TREEVERSE $f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \sigma\ \kappa\ \phi$
                                            in REST $f\ x\ \grave{z}\ \alpha\ y\ \delta\ (\tau - 1)\ \beta\ \sigma\ \kappa$
                                        else $\overleftarrow{\mathcal{J}}\ (\mathcal{I}\ f\ (\phi - \sigma))\ x\ \grave{y}$
REST $f\ x\ \grave{y}\ \alpha\ y\ \delta\ \tau\ \beta\ \sigma\ \phi$ = if $\phi - \sigma > \alpha \wedge \delta \neq 0 \wedge \tau \neq 0$
                                        then let $\kappa$ = MID $\delta\ \tau\ \sigma\ \phi$
                                                 $(\_, \grave{z})$ = TREEVERSE $f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \sigma\ \kappa\ \phi$
                                            in REST $f\ x\ \grave{z}\ \alpha\ y\ \delta\ (\tau - 1)\ \beta\ \sigma\ \kappa$
                                        else let $(\_, \grave{x}) = \overleftarrow{\mathcal{J}}\ (\mathcal{I}\ f\ (\phi - \sigma))\ x\ \grave{y}$
                                            in $(y, \grave{x})$

                         $\overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$ = let $n$ = PRIMOPS $f\ x$
                                              PICK $\alpha\ d\ t$
                                        in TREEVERSE $f\ x\ \grave{y}\ \alpha\ d\ t\ 0\ 0\ n$
```

Figure 26. Implementation of TREEVERSE from [9, Figure 4] using the general-purpose interruption and resumption interface, written in a functional style with no mutation. The variables δ, τ, β, σ, ϕ, n, d, and t have the same meaning as in [9]. The variables $f, x, \grave{x}, y, \grave{y}, z$, and $\grave{z}$ have the same meaning as earlier in this paper. The variable α denotes an upper bound on the number of evaluation steps for a leaf node. Different termination criteria allow the user to specify some of α, d, and t and compute the remainder as a function of the ones specified, together with n.

full functionality of that algorithm with the ability to select arbitrary execution points as split points. By selecting the choice of MID as either [9, Equation (12)] or [9, Equation (16)], one can select between bisection and binomial checkpointing. By selecting which of d, t, and α the user specifies, computing the others from the ones specified, together with n, using the methods described in [9] one can select the termination criterion to be either fixed space overhead, fixed time overhead, or logarithmic space and time overhead.[8] All of this functionality has been implemented in all three of our implementations: the interpreter, the hybrid compiler/interpreter, and the compiler.

But it turns out that the binary checkpointing algorithm from Figure 18 can be easily modified to support all of the functionality of the treeverse algorithm, including the ability to select either bisection or binomial checkpointing and the ability to select any of the termination criteria, including either fixed space overhead, fixed time overhead, or logarithmic space and time overhead, with exactly the same guarantees as treeverse. The idea is simple and follows from the observation that the right branch introduces a snapshot and the left branch introduces (re)computation of the primal. One maintains two counts, a right-branch count δ and a left-branch count τ, decrementing them as one descends into a right or left branch respectively, to limit the number of snapshots or the amount of (re)computation introduced. The base case is triggered when either gets to zero or a specified constant bound on the number of steps to be taped is reached. The binary checkpoint tree so produced corresponds to the associated n-ary checkpoint tree produced by

```
BINARY $f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \phi$ = if $\phi \le \alpha \vee \delta = 0 \vee \tau = 0$
                          then $\overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$
                          else let $\kappa$ = MID $\delta\ \tau\ 0\ \phi$
                                   $z$ = INTERRUPT $f\ x\ \kappa$
                                   $(y, \grave{z})$ = BINARY $\mathcal{R}\ z\ \grave{y}\ (\delta - 1)\ \tau\ (\phi - \kappa)$
                                   $(z, \grave{x})$ = BINARY $(\mathcal{I}\ f\ \kappa)\ x\ \grave{z}\ \delta\ (\tau - 1)\ \kappa$
                               in $(y, \grave{x})$

            $\overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$ = let $n$ = PRIMOPS $f\ x$
                              PICK $\alpha\ d\ t$
                          in BINARY $f\ x\ \grave{y}\ \alpha\ d\ t\ n$
```

Figure 27. Implementation of binary checkpointing using the general-purpose interruption and resumption interface in a fashion that supports all of the functionality of TREEVERSE from [9, Figure 4]. The variables δ, τ, ϕ, n, d, and t have the same meaning as in [9]. The variables f, x, $\grave{x}$, y, $\grave{y}$, z, and $\grave{z}$ have the same meaning as earlier in this paper. The variable α denotes an upper bound on the number of evaluation steps for a leaf node. Different termination criteria allow the user to specify some of α, d, and t and compute the remainder as a function of the ones specified, together with n.

treeverse, as discussed in Section 1. Again, this supports the full functionality of treeverse with the ability to select arbitrary execution points as split points. By selecting the choice of MID as either [9, Equation (12)] or [9, Equation (16)], one can select between bisection and binomial checkpointing. By selecting which of d, t, and α the user specifies, computing the others from the ones specified, together with n, using the methods described in [9] one can select the termination criterion to be either fixed space overhead, fixed time overhead, or logarithmic space and time overhead. All of this functionality has been implemented in all three of our implementations: the interpreter, the hybrid compiler/interpreter, and the compiler.

As per [9], with a binomial strategy for selecting split points, the termination criteria can be implemented as follows. Given a measured number n of evaluation steps, d, t, and α are mutually constrained by a single constraint

$$n = \textsc{primops}\ f\ x, \qquad \eta(d,t) = \binom{d+t}{t}, \qquad \alpha = \left\lceil \frac{n}{\eta(d,t)} \right\rceil \tag{23}$$

One can select any two and determine the third. Selecting d and α to determine $t = O(\sqrt[d]{n})$ yields the fixed space overhead termination criterion. Selecting t and α to determine $d = O(\sqrt[t]{n})$ yields the fixed time overhead termination criterion. Alternatively, one can further constrain $d = t$. With this, selecting α to determine d and t yields the logarithmic space and time overhead termination criterion.

6. An example

As discussed in Section 2, existing implementations of divide-and-conquer checkpointing, such as TAPENADE, are limited to placing split points at execution points corresponding to particular syntactic program points in the source code, i.e. loop iteration boundaries. Our approach can place split points at arbitrary execution points. The example in Figure 8 illustrates a situation where placing split points only at loop iteration boundaries can fail to

```
(define (car (cons car cdr)) car)

(define (cdr (cons car cdr)) cdr)

(define (first x) (car x))

(define (rest x) (cdr x))

(define (second x) (first (rest x)))

(define (iota n) (if (zero? n) '() (cons n (iota (- n 1)))))

(define (ilog2 l) (floor (/ (log l) (log 2))))

(define (rotate theta x1 x2)
 (let ((c (cos theta)) (s (sin theta)))
  (cons (- (* c x1) (* s x2)) (+ (* s x1) (* c x2)))))

(define (rot1 theta x)
 (if (or (null? x) (null? (rest x)))
     x
     (let ((x12 (rotate theta (first x) (second x))))
      (cons (car x12)
            (cons (cdr x12) (rot1 theta (rest (rest x))))))))

(define (rot2 theta x)
 (if (null? x) x (cons (first x) (rot1 theta (rest x)))))

(define (magsqr x)
 (if (null? x) 0 (+ (* (first x) (first x)) (magsqr (rest x)))))
```

```
(define (write-vector v)
 (if (null? v) '() (cons (write-real (first v)) (write-vector (rest v)))))

(define (f x l phi)
 (let outer ((i 1) (x1 x))
  (if (> i l)
      (/ (magsqr x1) 2)
      (let ((m (expt
                2
                (- (ilog2 l)
                   (ilog2
                    (+ 1 (modulo (* (* 1013 (floor (expt 3 phi))) i) l)))))))
       (let inner ((j 1) (x1 x1))
        (if (> j m)
            (outer (+ i 1) x1)
            (inner (+ j 1)
                   (let ((y (sqrt (magsqr x1))))
                    (rot2 (* 1.4 y) (rot1 (* 1.2 y) x1))))))))))

(let* ((n (read-real))
       (l (read-real))
       (phi (read-real))
       (x (iota n))
       (result (checkpoint-*j (lambda (x) (f x l phi)) x 1)))
 (cons (write-real (car result)) (write-vector (cdr result))))
```

Figure 28. A rendering of the example from Figure 8 in CHECKPOINTVLAD. Space and time overhead of two variants of this example when run under CHECKPOINTVLAD are presented in Figure 29. The variant for divide-and-conquer checkpointing is shown. The variant with no checkpointing replaces `checkpoint-*j` with `*j`.

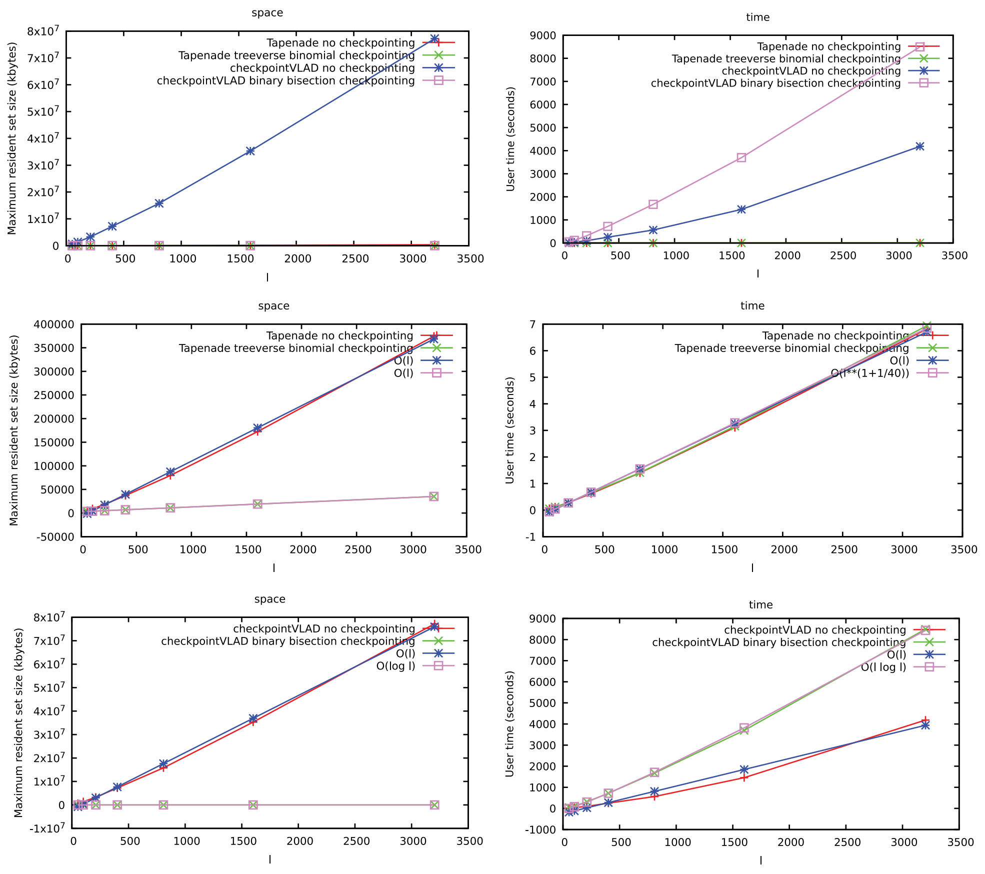

Figure 29. Space and time usage of reverse-mode AD with and without divide-and-conquer checkpointing for the example in Figures 8 and 28. Space and time usage was measured with `/usr/bin/time -verbose` The centre and bottom rows repeat the information from the top row just for TAPENADE and CHECKPOINTVLAD, overlaid with the theoretical asymptotic complexity fit to the actual data by linear regression.

yield the sublinear space overhead of divide-and-conquer checkpointing while placement of split points at arbitrary execution points will yield the sublinear space overhead of divide-and-conquer checkpointing. To illustrate this, we run the Fortran variant of this example two different ways with Tapenade:

(1) without checkpointing, by removing all pragmas and
(2) with divide-and-conquer checkpointing, particularly the treeverse algorithm applied to a root execution interval corresponding to the invocations of the outer DO loop, split points selected with the binomial criterion from execution points corresponding to iteration boundaries of the outer DO loop, and a fixed space overhead termination criterion, by placing the `c$ad binomial-ckp` pragma as shown in Figure 8.

For comparison, we reformulate this Fortran example in checkpointVLAD (Figure 28) and run it two different ways:

(1) without checkpointing, by calling $\overleftarrow{\mathcal{J}}$, written here as `*j` and
(2) with divide-and-conquer checkpointing, particularly the binary checkpointing algorithm applied to a root execution interval corresponding to the entire derivative calculation, split points selected with the bisection criterion from arbitrary execution points, and a logarithmic space and time overhead termination criterion, by calling $\overset{\checkmark}{\mathcal{J}}$, written here as `checkpoint-*j`.

For this example, n is the input dimension and l is the number of iterations of the outer loop. Using the notation from Section 4, the maximal space usage of the primal for this example should be $w = O(n)$. The time required for the primal for this example should be $t = O(l)$, since there are l iterations of the outer loop and the inner loop has average case $O(1)$ iterations per iteration of the outer loop. The analysis in Sections 4 and 5 predicts the following asymptotic space and time complexity of the Tapenade and checkpointVLAD variants that compute gradients on this particular example:

Computation	Complexity	
	space	time
primal	$O(n)$	$O(l)$
Tapenade no checkpointing	$O(n+l)$	$O(l)$
Tapenade divide-and-conquer checkpointing	$O(nl)$[9]	$O(l\sqrt[d]{l})$
checkpointVLAD no checkpointing	$O(n+l)$	$O(l)$
checkpointVLAD divide-and-conquer checkpointing	$O(n \log l)$	$O(l \log l)$

The efficacy of our method can be seen in the plots (Figure 29) of the observed space and time usage of the above two Fortran variants and the above two checkpointVLAD variants with varying l and $n = 1000$. We observe that Tapenade space and time usage grows with l for all cases. checkpointVLAD space and time usage grows with l with $\overleftarrow{\mathcal{J}}$. checkpointVLAD space usage is sublinear with $\overset{\checkmark}{\mathcal{J}}$. checkpointVLAD time usage grows with l with $\overset{\checkmark}{\mathcal{J}}$.

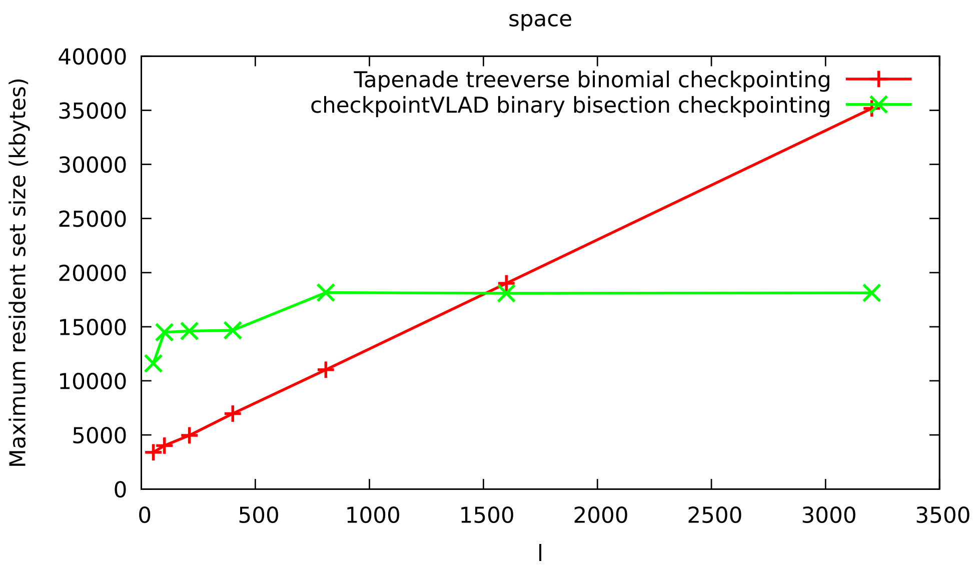

Figure 30. Comparison of space usage of the the example in Figures 8 and 28 when run with divide-and-conquer checkpointing. CHECKPOINTVLAD achieves sublinear growth while TAPENADE does not, thus for long enough run times, the space usage of TAPENADE exceeds that of CHECKPOINTVLAD.

The crucial aspect of this example is that we observe sublinear space usage overhead with divide-and-conquer checkpointing in CHECKPOINTVLAD but not in TAPENADE (Figure 30).[10] The reason that we fail to observe sublinear space usage overhead with divide-and-conquer checkpointing in TAPENADE is that the space overhead guarantees only hold when the asymptotic time complexity of the loop body is constant. Since the asymptotic time complexity of the loop body is $O(l)$, the requisite tape size grows with $O(l)$ even though the number of snapshots, and the size of those snapshots, is bounded by a constant.

7. Discussion

Our current implementations, as evaluated in Section 6, are expository prototypes and not intended as practical artifacts. Nonetheless, technology exists that can support construction of large-scale practical and efficient implementations based on the conceptual ideas presented here.

7.1. Implementation technologies

One can use POSIX `fork()` to implement the general-purpose interruption and resumption interface, allowing it to apply in the host, rather than the target, and thus it could be used to provide an overloaded implementation of divide-and-conquer checkpointing in a fashion that was largely transparent to the user [9]. The last paragraph of [9] states:

> For the sake of user convenience and computational efficiency, it would be ideal if reverse automatic differentiation were implemented at the compiler level.

We have exhibited such a compiler. Our implementation, however, is not as efficient as Tapenade. There can be a number of reasons for this. First, Fortran unboxes double precision numbers whereas checkpointVLAD boxes them. This introduces storage allocation, reclamation, and access overhead for arithmetic operations. Second, array access and update in Fortran take constant time whereas access and update in checkpointVLAD take linear time. Array update in checkpointVLAD further involves storage allocation and reclamation overhead. Third, Tapenade implements the base case reverse mode of divide-and-conquer checkpointing using source-code transformation whereas checkpointVLAD implements it with operator overloading. In particular, the dynamic method for supporting nesting involves tag dispatch for every arithmetic operation.

We have exhibited a very aggressive compiler (Stalin∇) for vlad that ameliorates some of these issues. It unboxes double precision numbers and implements AD via source-code transformation instead of operator overloading as is done by checkpointVLAD. This allows it to have numerical performance rivaling Fortran. While it does not support constant-time array access and update, methods that are well-known in the programming languages community (e.g. monads and uniqueness types) can be used for that. But it does not include support for divide-and-conquer checkpointing. However, there is no barrier, in principle, to supporting divide-and-conquer checkpointing, of the sort described above, in an aggressive optimizing compiler that implements AD via source-code transformation. One would simply need to reformulate the source-code transformations that implement AD, along with the aggressive compiler optimizations, in CPS instead of direct style. Moreover, the techniques presented here could be integrated into other compilers for other languages that generate target code in CPS by instrumenting the CPS conversion with step counting, step limits, and limit-check interruptions.[11] A driver can be wrapped around such code to implement $\overset{\checkmark}{\mathcal{J}}$. For example, existing compilers, like sml/nj [4], for functional languages like sml, already generate target code in CPS, so it seems feasible to adapt their techniques to the purpose of AD with divide-and-conquer checkpointing. In fact, the overhead of the requisite instrumentation for step counting, step limits, and limit-check interruptions need not be onerous because the step counting, step limits, and limit-check 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 [27], for languages like sml, that achieve very low overhead for automatic storage management.

7.2. Advantages of functional languages for interruption and resumption

Functional languages simplify interruption and resumption, allowing these to be much more efficient. Two different capsules taken at two different execution points can share common substructure, by way of pointers, without needing to copy that substructure. Indeed, CPS in the checkpointVLAD implementation renders all program state, including the stack and variables in the environment, as closures, possibly nested. Creating a capsule simply involves saving a pointer to a closure. Resuming a capsule simply involves invoking the saved closure, a simple function call that is passed the closure environment as its argument. The garbage collector can traverse the pointer structure of the program state to determine the lifetime of a capsule. The interruption and resumption framework need not do so itself. This simplicity and efficiency would be disrupted by structure mutation or assignment.

7.3. Nesting of AD operators

It has been argued that the ability to nest AD operators is important in many practical domains [2,3,7]. Supporting nested use of AD operators involves many subtle issues [19]. CHECKPOINTVLAD addresses these issues and fully supports nested use of AD operators. One can write programs of the form

$$\alpha\ (\lambda x.(\ldots(\beta\, f\ \ldots)))\ \ldots, \tag{24}$$

where each of α and β can be any of $\overrightarrow{\mathcal{J}}$, $\overleftarrow{\mathcal{J}}$, and $\overset{\checkmark}{\mathcal{J}}$. I.e. one can apply AD to one function that, in turn, applies AD to another function. This allows one not only to do forward-over-reverse, reverse-over-forward, and reverse-over-reverse, it also allows one to do things like divide-and-conquer-reverse-over-forward, reverse-over-divide-and-conquer-reverse, and even divide-and-conquer-reverse-over-divide-and-conquer-reverse.

There is one catch however. When one applies an AD operator, that application is considered to be atomic by an application of a surrounding AD operator. This is evident by the $n+1$ in (5l, 5m, 6r), What this means is that if α in (24) were $\overset{\checkmark}{\mathcal{J}}$, a split point for α could not occur inside *f*. While this does not affect the correctness of the result, it could affect the space and time complexity.

The reason for this is that while the semantics of the AD operators are functional, their internal implementation involves mutation. In particular, to support nesting, $\overrightarrow{\mathcal{J}}$, $\overleftarrow{\mathcal{J}}$, and $\overset{\checkmark}{\mathcal{J}}$ internally maintain and update an ϵ tag as described in [19]. The ϵ tag is incremented upon entry to an AD operator and decremented upon exit from that invocation to keep track of the nesting level. All computation within a level must be performed with the same ϵ tag. This requires that the entries to and exits from AD operator invocations obey last-in-first-out sequencing. If α were $\overset{\checkmark}{\mathcal{J}}$, and the computation of f were interrupted, then situations could arise where the last-in-first-out sequencing of β was violated. Moreover, since divide-and-conquer checkpointing executes different portions of the forward sweep different numbers of times, the number of entries into a nested AD operator could exceed the number of exits from that operator.

Reverse mode involves a further kind of mutation. The forward sweep creates a tape represented as a directed acyclic graph. The nodes in this tape contain slots for the cotangent values associated with the corresponding primal values. The reverse sweep operates by traversing this graph to accumulate the cotangents in these slots. Such accumulation is done by mutation.

The above issues arise because of mutation in the implementation of AD operators. Conceivably, these could be addressed using methods that are well-known in the programming languages community for supporting mutation in functional languages (e.g. monads and uniqueness types). Issues arise beyond this, however. If both α and β were $\overset{\checkmark}{\mathcal{J}}$, and $\overset{\checkmark}{\mathcal{J}}$ was not atomic, situations could arise where f could interrupt for α instead of β. Currently, interruption is indicated by returning instead of calling a continuation. If f were to return instead of calling a continuation, there is no way to indicate that that interruption was due to α instead of β. It is unclear whether this issue could be resolved.

8. Conclusion

Reverse-mode AD with divide-and-conquer checkpointing is an enabling technology, allowing gradients to be efficiently calculated even where classical reverse mode imposes an impractical storage overhead. We have shown that it is possible to provide an operator that implements reverse-mode AD with divide-and-conquer checkpointing, implemented as an interpreter, a hybrid compiler/interpreter, and a compiler, which

- has an identical API to the classical reverse-mode AD operator,
- requires no user annotation,
- takes the entire derivative calculation as the root execution interval, not just the execution intervals corresponding to the invocations of particular constructs such as DO loops,
- takes arbitrary execution points as candidate split points, not just the execution points corresponding to the program points at the boundaries of particular constructs like the iteration boundaries of DO loops,
- supports both an algorithm that constructs binary checkpoint trees and the treeverse algorithm that constructs n-ary checkpoint trees,
- supports selection of actual split points from candidate split points using both a bisection and a binomial criterion, and
- supports any of the termination criteria of fixed space overhead, fixed time overhead, or logarithmic space and time overhead,

yet still provides the favourable storage requirements of reverse mode with divide-and-conquer checkpointing, guaranteeing sublinear space and time overhead.

Notes

1. The distinction between gold and pink lines, the meaning of brown lines, and the meaning of the black tick marks on the left of the gold and pink lines will be explained in Section 3.10.
2. The correspondence between capsules and snapshots will be discussed in Section 3.10.
3. The surface syntax employed differs slightly from SCHEME in ways that are irrelevant to the issue at hand.
4. In the implementation, $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ are named `j*` and `*j`, respectively.
5. https://github.com/qobi/R6RS-AD and https://engineering.purdue.edu/~qobi/stalingrad-examples2009/
6. http://diffsharp.github.io/DiffSharp/
7. In the implementation, $\overset{\checkmark}{\mathcal{J}}$ is named `checkpoint-*j`.
8. In Section 5 and Figures 26 and 27 we use notation similar to that in [9] to facilitate understanding. Thus n and t here means something different then elsewhere in this paper.
9. The space complexity of TAPENADE with divide-and-conquer checkpointing would be $O(n)$ if the inner DO loop would have a constant number of iterations. The fact that it has average case $O(1)$ iterations but $O(l)$ worst case, foils checkpointing and causes the space complexity to increase.

10. Technically, the space and time usage overhead of the CHECKPOINTVLAD variant of this example with divide-and-conquer checkpointing should be logarithmic. It is difficult to see that precise overhead in the plots. Linear regression does indeed fit logarithmic growth to the time usage better than linear growth. But the observed space usage appears to be grow in steps. This is likely due to the coarse granularity of the measurement techniques that are based on kernel memory page allocation and thus fail to measure the actual live fraction of the heap data managed by the garbage collector.
11. We note that many optimizing compilers, for example GCC, use an intermediate program representation called Single Static Assignment, or SSA, which is formally equivalent to CPS [16].

Acknowledgements

This paper is an expanded version of the extended abstract of a presentation at AD (2016), which is available as a technical report [20]. Aspects of this work may be subject to the intellectual property policies of Purdue University and/or Maynooth University. Code for the implementations described above and related materials are available at https://github.com/qobi/checkpoint-VLAD/. Any opinions, findings, views, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, official policies, or endorsements, either expressed or implied, of the sponsors. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported, in part, by the US National Science Foundation under Grants 1522954-IIS and 1734938-IIS, by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior/Interior Business Center (DOI/IBC) contract number D17PC00341, and by Science Foundation Ireland under Grant SFI 09/IN.1/I2637.

ORCID

Jeffrey Mark Siskind http://orcid.org/0000-0002-0105-6503
Barak A. Pearlmutter http://orcid.org/0000-0003-0521-4553

References

[1] P. Achten, J. Van Groningen, and R. Plasmeijer, *High level specification of I/O in functional languages*, Functional Programming, Glasgow 1992, Springer, Ayr Scotland, 1993, pp. 1–17.
[2] N. Agarwal, B. Bullins, and E. Hazan, *Second order stochastic optimization in linear time*, 2016. Available at https://arxiv.org/abs/1602.03943.
[3] M. Andrychowicz, M. Denil, S.G. Colmenarejo, M.W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, *Learning to learn by gradient descent by gradient descent*, in *Neural Information Processing Systems*, NIPS, 2016. Available at https://arxiv.org/abs/1606.04474.
[4] A.W. Appel, *Compiling with Continuations*, Cambridge University Press, Cambridge, UK, 2006.
[5] 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, 1996.
[6] T. Chen, B. Xu, Z. Zhang, and C. Guestrin, *Training deep nets with sublinear memory cost*, 2016. Available at https://arxiv.org/abs/1604.06174.

[7] B. Christianson, *Reverse accumulation of functions containing gradients*, Tech. Rep. 278, University of Hertfordshire Numerical Optimisation Centre, presented at the Theory Institute Argonne National Laboratory Illinois, Procs. of the Theory Institute on Combinatorial Challenges in Automatic Differentiation, 1993. Available at http://hdl.handle.net/2299/4337.

[8] B. Dauvergne and L. Hascoët, *The data-flow equations of checkpointing in reverse automatic differentiation*, in *Computational Science – ICCS 2006*, Lecture Notes in Computer Science, Vol. 3994, V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, and J.J. Dongarra, eds., Springer, Heidelberg, 2006, pp. 566–573.

[9] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optim. Methods Softw. 1 (1992), pp. 35–54.

[10] A. Griewank, D. Juedes, and J. Utke, *A package for the automatic differentiation of algorithms written in C/C++. User manual*, Tech. Rep., Institute of Scientific Computing, Technical University of Dresden, Dresden, Germany, 1996. Available at http://www.math.tu-dresden.de/adol-c/adolc110.ps.

[11] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-efficient backpropagation through time*, in *Neural Information Processing Systems*, NIPS, 2016. Available at https://arxiv.org/abs/1606.03401.

[12] L. Hascoët and V. Pascual, *TAPENADE 2.1 user's guide*, Rapport technique 300, INRIA, 2004.

[13] C.T. Haynes and D.P. Friedman, *Engines build process abstractions*, Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, ACM, Austin, TX, 1984, pp. 18–24.

[14] R. Heller, *Checkpointing without operating system intervention: Implementing Griewank's algorithm*, Master's thesis, Ohio University, 1998.

[15] Y. Kang, *Implementation of forward and reverse mode automatic differentiation for GNU Octave applications*, Master's thesis, Ohio University, 2003.

[16] R.A. Kelsey, *A correspondence between continuation passing style and static single assignment form*, ACM SIGPLAN Notices, Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, vol. 30, 1995, pp. 13–22.

[17] A. Kowarz and A. Walther, *Optimal checkpointing for time-stepping procedures in ADOL-C*, in *Computational Science – ICCS 2006*, Lecture Notes in Computer Science, Vol. 3994, V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, and J.J. Dongarra, eds., Springer, Heidelberg, 2006, pp. 566–573.

[18] J.C. Reynolds, *The discoveries of continuations*, Lisp Symb. Comput. 6 (1993), pp. 233–247.

[19] J.M. Siskind and B.A. Pearlmutter, *Nesting forward-mode AD in a functional framework*, Higher-Order Symb. Comput. 21 (2008), pp. 361–376.

[20] J.M. Siskind and B.A. Pearlmutter, *Binomial checkpointing for arbitrary programs with no user annotation*, 2016. Available at https://arxiv.org/abs/1611.03410.

[21] J.M. Siskind and B.A. Pearlmutter, *Efficient implementation of a higher-order language with built-in AD*, 2016. Available at https://arxiv.org/abs/1611.03416.

[22] B. Speelpenning, *Compiling fast partial derivatives of functions given by algorithms*, Ph.D. diss., Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

[23] A. Stovboun, *A tool for creating high-speed, memory efficient derivative codes for large scale applications*, Master's thesis, Ohio University, 2000.

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

[25] Y.M. Volin and G.M. Ostrovskii, *Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis*, Comput. Math. Appl. 11 (1985), pp. 1099–1114.

[26] P.L. Wadler, *Comprehending monads*, Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, ACM, Nice, France, 1990, pp. 61–78.

[27] S. Weeks, *Whole-program compilation in MLton, Workshop on ML*, 2006. Available at http://www.mlton.org/References.attachments/060916-mlton.pdf.