# 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, 465 Northwestern Avenue, West Lafayette, IN, USA;* [b] *Dept of Computer Science, Maynooth University, Ireland*

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.

**Keywords:** Reverse-Mode Automatic Differentiation; Binomial Checkpointing; Treeverse; Programming Language Theory; Compiler Theory; Lambda Calculus

*AMS Subject Classification*: 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 [37]. 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* [42] 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 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.

*Corresponding author. Email: qobi@purdue.edu

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.

Figs. 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 $p_0, \ldots, p_6$ 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 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 Figs. 1 and 2 other than Fig. 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.

Fig. 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 $x$. Such is performed without taping (green). Fig. 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 $O(t)$. Fig. 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

[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.

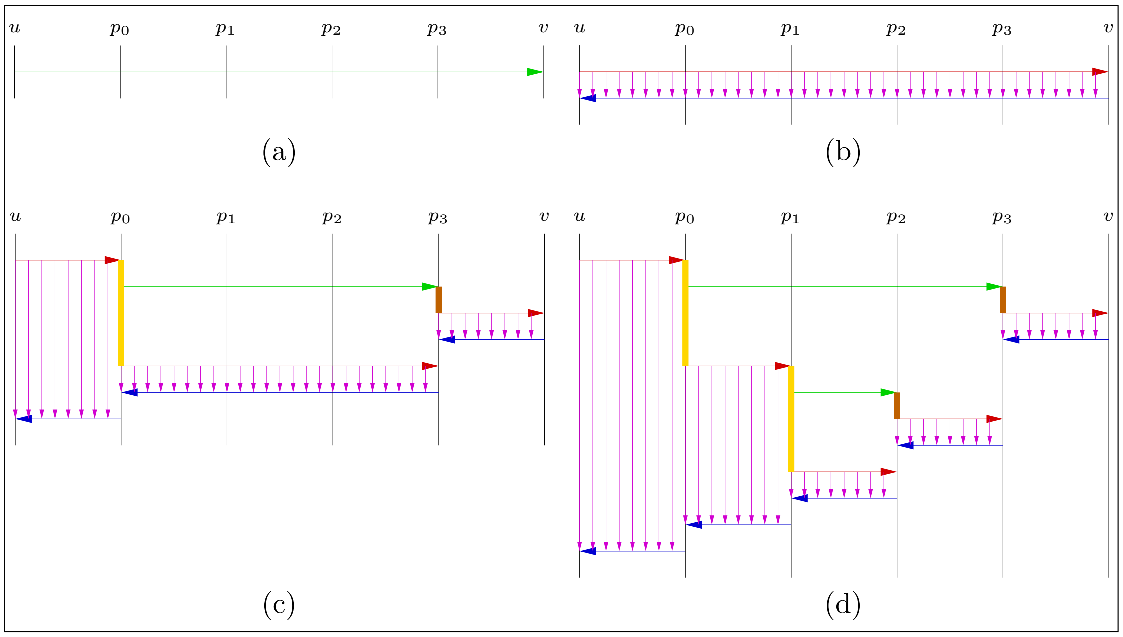


Figure 1. Checkpointing in reverse-mode AD. See text for description.

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 such 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 Fig. 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 [42].

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 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 [17] 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 [11]. 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 sweep into stages run sequentially [13, 15]. 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 Fig. 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 Fig. 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 $[u, v)$. 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*

$$\begin{tikzcd}
& {[u,v)} \arrow[dl, no head] \arrow[dr, no head] & \\
{[u,p)} & & {[p,v)}
\end{tikzcd}$$

whose nodes are labeled with execution intervals, the intervals of the children of a node are adjacent, the interval of 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 $p_2$.

$$\begin{tikzcd}
& & & {[u,v)} \arrow[dl, no head] \arrow[dr, no head] & \\
& & {[u,p_2)} \arrow[dl, no head] \arrow[dr, no head] & & {[p_2,v)} \\
& {[u,p_1)} \arrow[dl, no head] \arrow[dr, no head] & & {[p_1,p_2)} & \\
{[u,p_0)} & & {[p_0,p_1)} & &
\end{tikzcd}$$

This can also be viewed as constructing an *n-ary checkpoint tree*

$$\begin{tikzcd}
& {[u,v)} \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] \arrow[drr, no head] & & \\
{[u,p_0)} & {[p_0,p_1)} & {[p_1,p_2)} & {[p_2,v)}
\end{tikzcd}$$

where all children but the rightmost are checkpointed. This leads to *nested* checkpoints for the execution intervals $[u, p_0)$, $[u, p_1)$, and $[u, p_2)$ as shown in Fig. 2(c). Since the starting

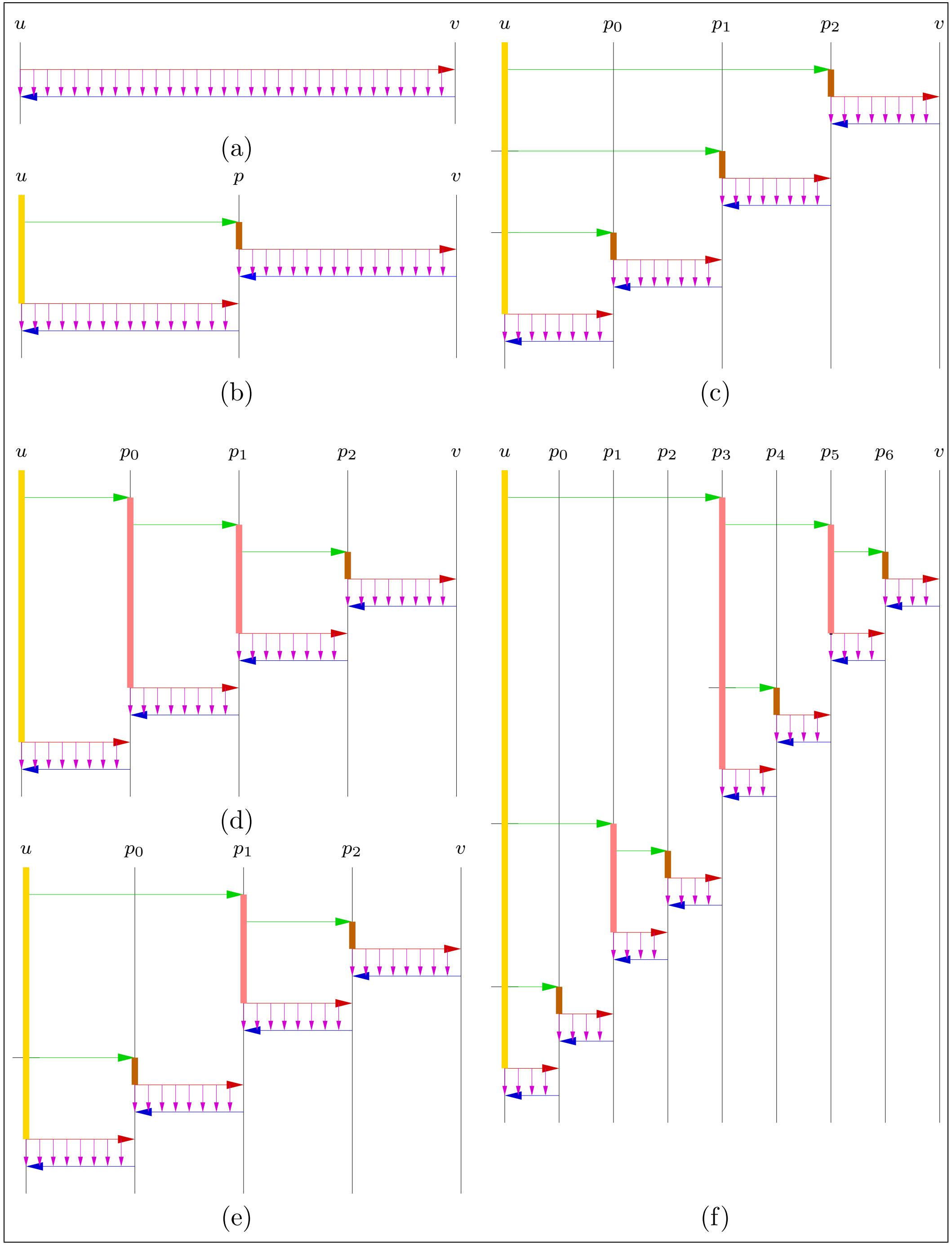


Figure 2. Divide-and-conquer checkpointing in reverse-mode AD. See text for description.

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 $O(t)$ 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 $p_2$.

$$\begin{tikzcd}
 & {[u,v)} \arrow[dl, no head] \arrow[dr, no head] & & & \\
{[u,p_0)} & & {[p_0,v)} \arrow[dl, no head] \arrow[dr, no head] & & \\
 & {[p_0,p_1)} & & {[p_1,v)} \arrow[dl, no head] \arrow[dr, no head] & \\
 & & {[p_1,p_2)} & & {[p_2,v)}
\end{tikzcd}$$

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 arbitrarily while running the primal (green) just once, by saving more snapshots (gold and pink), as shown in Fig. 2(d), This requires $O(t)$ 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 $O(t)$ increase in time for (re)computation of the primal (green). The right-branching binary checkpoint trees require $O(t)$ increase in space for storage of the live snapshots (gold and pink) but (re)run the primal only once.

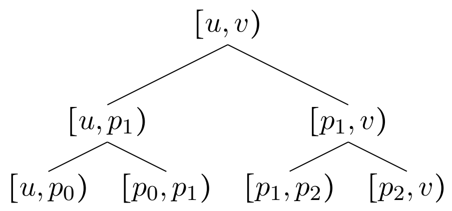
One can also construct a complete binary checkpoint tree over the same root execution interval $[u, v)$ with the same split points $p_0$, $p_1$, and $p_2$.

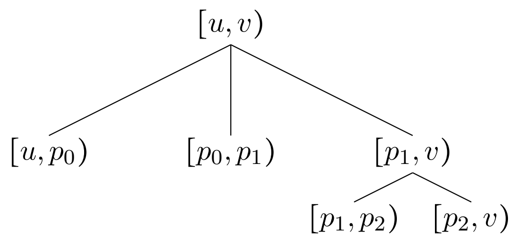
$$\begin{tikzcd}
 & & & {[u,v)} \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & {[u,p_1)} \arrow[dl, no head] \arrow[dr, no head] & & & & {[p_1,v)} \arrow[dl, no head] \arrow[dr, no head] & \\
{[u,p_0)} & & {[p_0,p_1)} & & {[p_1,p_2)} & & {[p_2,v)}
\end{tikzcd}$$

This constitutes application of the approach from Fig. 2(b) in a divide-and-conquer fashion as shown in Fig. 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 Fig. 2(f). This leads to $O(\log t)$ increase in space for storage of the live snapshots (gold and pink) and $O(\log t)$ 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]

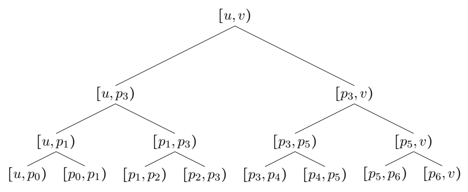
Any given divide-and-conquer decomposition of the same root execution interval with the same split points can be viewed as either a binary checkpoint tree or an n-ary checkpoint tree. Thus Fig. 2(e) can be viewed as either of the following.

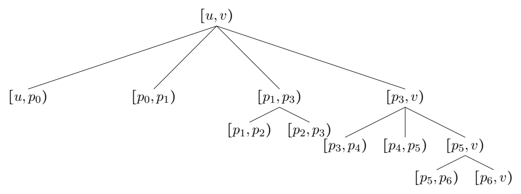
[2]The correspondence between capsules and snapshots will be discussed in Section 3.10.





Similarly, Fig. 2(f) can be viewed as either of the following.





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 [13, Figs. 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 [13, equation (12)]. One can employ a variety of termination criteria, including that from [13, 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 [13, equation (16)] The termination criterion from [13, p. 46] is usually adopted to achieve the desired properties discussed in [13]. 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 [14] contains a nested taping mechanism for time-integration processes [21] 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 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 two examples.

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 because such would lead to a right-branching checkpoint tree and behavior analogous to Fig. 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.

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 [13]. 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 [18, 19, 39] 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 Listing 1. This example, $y = f(x)$, while contrived, is a simple caricature of a situation that arises commonly in practice, *e.g.*, in adaptive grid methods. Here, the duration of the inner loop varies wildly as some function $l(x, i)$ of the input and the outer loop index, perhaps $2^{\lfloor \lg(n) \rfloor - \lfloor \lg(1 + (1007 \lfloor 3^x \rfloor i \bmod n)) \rfloor}$, that is small on most iterations of the outer loop but $O(n)$ on a few iterations. 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 such. 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 now

Listing 1 FORTRAN example. This example is rendered in CHECKPOINTVLAD in Listing 2. Space and time overhead of two variants of this example when run under TAPENADE are presented in Fig. 22. 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.

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

      subroutine f(n, x, y)
      y = x
c$ad binomial-ckp n+1 30 1
      do i = 1, n
         m = 2**(ilog2(n)-
     +           ilog2(1+int(mod(real(x, 8)**3*real(i, 8)*
     +                                             1007.0d0,
     +                                  real(n, 8)))))
         do j = 1, m
            y = y*y
            y = sqrt(y)
         end do
      end do
      end

      program main
      read *, n
      read *, x
      read *, yb
      call f(n, x, y)
      call f_b(n, x, xb, y, yb)
      print *, y
      print *, xb
      end
```

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 7.2). 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

$$\mathcal{A}\ \langle(\lambda x.e),\rho\rangle\ v = \mathcal{E}\ \rho[x \mapsto v]\ e \tag{2}$$
$$\mathcal{E}\ \rho\ c = c \tag{3}$$
$$\mathcal{E}\ \rho\ x = \rho\ x \tag{4}$$
$$\mathcal{E}\ \rho\ (\lambda x.e) = \langle(\lambda x.e),\rho\rangle \tag{5}$$
$$\mathcal{E}\ \rho\ (e_1\ e_2) = \mathcal{A}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2) \tag{6}$$
$$\mathcal{E}\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \textbf{if}\ (\mathcal{E}\ \rho\ e_1)\ \textbf{then}\ (\mathcal{E}\ \rho\ e_2)\ \textbf{else}\ (\mathcal{E}\ \rho\ e_3) \tag{7}$$
$$\mathcal{E}\ \rho\ (\diamond e) = \diamond(\mathcal{E}\ \rho\ e) \tag{8}$$
$$\mathcal{E}\ \rho\ (e_1 \bullet e_2) = (\mathcal{E}\ \rho\ e_1) \bullet (\mathcal{E}\ \rho\ e_2) \tag{9}$$

Figure 3. Direct-style evaluator for the core CHECKPOINTVLAD language.

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 [26, 27, 34]. Support for AD in general, and divide-and-conquer checkpointing in particular, is simplified in a functional programming language (see Section 7.3). Except for this simplification, which can be eliminated with well-known techniques (*e.g.*, monads [43] 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 7.2).

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 (Fig. 3) and extend such 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 [30]. 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.

With any evaluator, one distinguishes between two language evaluation strata: the *target*, the language being implemented and the process of evaluating programs in that

[3]The surface syntax employed differs slightly from SCHEME in ways that are irrelevant to the issue at hand.

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 Fig. 3, $\rho$ denotes an *environment*, a mapping from variables to their values, $\rho_0$ denotes the empty environment that does not map any variables, $\rho\,x$ denotes looking up the variable $x$ in the environment $\rho$ to obtain its value, $\rho[x \mapsto v]$ denotes augmenting an environment $\rho$ 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 $\rho$. There is a clause for $\mathcal{E}$ in Fig. 3, (3) to (9), for each construct in (1). Clause (3) says that one evaluates a constant by returning that constant. Clause (4) 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 $\rho$ containing values for the free variables in $e$. Clause (5) 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 (6) 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 (2), 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 (9), 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 [26, 27, 34]. 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 [32]. 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.

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

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

[4]In the implementation, $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ are named `j*` and `*j` respectively.

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \textbf{let}\ (v_4 \rhd \acute{v}_5) = (\mathcal{A}\ v_1\ (v_2 \rhd \acute{v}_3))\ \textbf{in}\ (v_4, \acute{v}_5) \tag{11}$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \textbf{let}\ (v_4 \lhd \grave{v}_5) = ((\mathcal{A}\ v_1\ v_2) \lhd \grave{v}_3)\ \textbf{in}\ (v_4, \grave{v}_5) \tag{12}$$

$$\mathcal{E}\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \overrightarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3) \tag{13}$$

$$\mathcal{E}\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \overleftarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3) \tag{14}$$

Figure 4. Additions to the direct-style evaluator for CHECKPOINTVLAD to support AD.

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∇ [36] 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 7.2).

In CHECKPOINTVLAD, AD is performed by overloading the arithmetic operations in the host, in a fashion similar to FADBAD++ [7]. 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 [32, 41]. 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.

$$e ::= \overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3 \mid \overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3 \tag{10}$$

We implement this functionality by augmenting the direct-style evaluator with new clauses for $\mathcal{E}$ (Fig. 4), clause (13) for $\overrightarrow{\mathcal{J}}$ and clause (14) for $\overleftarrow{\mathcal{J}}$. These clauses are all analogous to clause (9), formulated around $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators in the host. These are defined in (11) and (12). 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 $x$, $(\overrightarrow{\mathcal{J}}\ f\ x\ \acute{x})$ 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 (11), $\overrightarrow{\mathcal{J}}$ 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

[5]https://github.com/qobi/R6RS-AD and https://engineering.purdue.edu/~qobi/stalingrad-examples2009/
[6]http://diffsharp.github.io/DiffSharp/

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 $v_2 \triangleright \acute{v}_3$. $\mathcal{A}$ is then used to apply the function (closure) $v_1$ to the data structure produced by $v_2 \triangleright \acute{v}_3$. 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 **let** $(v_4 \triangleright \acute{v}_5) = \ldots$ **in** $\ldots$.

As described in (12), $\overleftarrow{\mathcal{J}}$ 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 $((\mathcal{A}\ v_1\ v_2) \triangleleft \grave{v}_3)$. 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 **let** $(v_4 \triangleright \grave{v}_5) = \ldots$ **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 below, 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 behavior of $\overset{\checkmark}{\mathcal{J}}$ is shown in Fig. 5. In this inductive definition, a function $f$ is split into the composition of two functions $g$ and $h$ in step (1), the capsule $z$ is computed by applying $g$ to the input $x$ in step (2), and the cotangent is computed by recursively applying $\overset{\checkmark}{\mathcal{J}}$ to $h$ and $g$ in steps 3 and 4. This divide-and-conquer behavior is terminated in a base case, when the function $f$ is small, at which point the cotangent is computed with $\overleftarrow{\mathcal{J}}$, in step (0). If step (1) splits a function $f$ into two functions $g$ and $h$ that take the same number of 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.

[7]In the implementation, $\overset{\checkmark}{\mathcal{J}}$ is named `checkpoint-*j`.

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

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

Figure 5. 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 6. General-purpose interruption and resumption interface.

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 nonlocal 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 (Fig. 6). 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 7.2.

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 Fig. 7. The function $f$ is split into the composition of two functions $g$ and $h$ by taking $g$ as $(\lambda x.\text{INTERRUPT}\ f\ x\ l)$, where $l$ is half the number of steps determined by $(\text{PRIMOPS}\ f\ x)$, and $h$ as $(\lambda z.\text{RESUME}\ z)$.

## 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) [30, 38, 40], where functions

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

**base case:** $$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y} \tag{0}$$

**inductive case:** $$l = \text{PRIMOPS}\ f\ x \tag{1}$$

$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$

$$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y} \tag{3}$$

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor)\ x\ \grave{z} \tag{4}$$

Figure 7. 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 $\lfloor \frac{l}{2} \rfloor$ and step (3) taking $l$ at the next recursion level to be $\lceil \frac{l}{2} \rceil$. As discussed in the text, this implementation is not quite correct, because $(\lambda z.\text{RESUME}\ z)$ in step (3) and $(\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor)$ in step (4) are host closures but need to be target closures. A proper implementation is given in Fig. 12.

(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 (Figs. 8 and 9). In such a style, 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.

Fig. 8 contains the portion of the CPS evaluator for the core language corresponding to Fig. 3, while Fig. 9 contains the portion of the CPS evaluator for the AD constructs corresponding to Fig. 4. Except for (16), the equations in Figs. 3 and 4 are in one-to-one correspondence to those in Figs. 8 and 9, in order. Clauses (17)–(19) are analogous to the corresponding clauses (3)–(5) 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 variants of

$$\begin{array}{l}\mathcal{E}\ (\lambda n\ l\ v_1.\\ \quad(\mathcal{E}\ (\lambda n\ l\ v_2.\\ \qquad\quad(k\ n\ l\ \ldots))\\ \qquad n\ l\ \rho\ e_2))\\ \quad(n+1)\ l\ \rho\ e_1\end{array} \tag{28}$$

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 (16) of $\mathcal{E}$ implements interrup-

$$\mathcal{A}\;k\;n\;l\;\langle(\lambda x.e),\rho\rangle\;v = \mathcal{E}\;k\;n\;l\;\rho[x\mapsto v]\;e \tag{15}$$

$$\mathcal{E}\;k\;l\;l\;\rho\;e = [\![k,\langle(\lambda\_.e),\rho\rangle]\!] \tag{16}$$

$$\mathcal{E}\;k\;n\;l\;\rho\;c = k\;(n+1)\;l\;c \tag{17}$$

$$\mathcal{E}\;k\;n\;l\;\rho\;x = k\;(n+1)\;l\;(\rho\;x) \tag{18}$$

$$\mathcal{E}\;k\;n\;l\;\rho\;(\lambda x.e) = k\;(n+1)\;l\;\langle(\lambda x.e),\rho\rangle \tag{19}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(e_1\;e_2) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\&\qquad\quad(\mathcal{A}\;k\;n\;l\;v_1\;v_2))\\&\qquad n\;l\;\rho\;e_2))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{20}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(\textbf{if}\;e_1\;\textbf{then}\;e_2\;\textbf{else}\;e_3) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\textbf{if}\;v_1\\&\quad\;\textbf{then}\;(\mathcal{E}\;k\;n\;l\;\rho\;e_2)\\&\quad\;\textbf{else}\;(\mathcal{E}\;k\;n\;l\;\rho\;e_3)))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{21}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(\diamond e) = \mathcal{E}\;&(\lambda n\;l\;v.\\&\quad(k\;n\;l\;(\diamond v)))\\&(n+1)\;l\;\rho\;e\end{aligned} \tag{22}$$

$$\begin{aligned}\mathcal{E}\;k\;n\;l\;\rho\;(e_1\bullet e_2) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\&\qquad\quad(k\;n\;l\;(v_1\bullet v_2)))\\&\qquad n\;l\;\rho\;e_2))\\&(n+1)\;l\;\rho\;e_1\end{aligned} \tag{23}$$

Figure 8. CPS evaluator for the core CHECKPOINTVLAD language.

tion. 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 $\rho$ 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 (28), 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 (22) and (23), as typified by (28), 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 (26) and (27), again as typified by (28), increment the step count by one. Fourth, like all builtin operators, $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ in the host, in (24) and (25), are implemented in direct style and are not passed a continuation. This means that clauses (26) and (27), as typified by (28), 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 (24) and (25) must be provided with a continuation $k$, a step count $n$,

$$\begin{aligned}\overrightarrow{\mathcal{J}}\; v_1\; v_2\; \acute{v}_3 = \mathcal{A}\; &(\lambda n\; l\; (v_4 \rhd \acute{v}_5).\\ &\quad (v_4, \acute{v}_5))\\ &\;0\; \infty\; v_1\; (v_2 \rhd \acute{v}_3)\end{aligned}\tag{24}$$

$$\begin{aligned}\overleftarrow{\mathcal{J}}\; v_1\; v_2\; \grave{v}_3 = \mathcal{A}\; &(\lambda n\; l\; v.\\ &\quad \mathbf{let}\; (v_4 \lhd \grave{v}_5) = v \lhd \grave{v}_3\\ &\quad \mathbf{in}\; (v_4, \grave{v}_5))\\ &\;0\; \infty\; v_1\; v_2\end{aligned}\tag{25}$$

$$\begin{aligned}\mathcal{E}\; k\; n\; l\; \rho\; (\overrightarrow{\mathcal{J}}\; e_1\; e_2\; e_3) = \mathcal{E}\; &(\lambda n\; l\; v_1.\\ &\quad (\mathcal{E}\; (\lambda n\; l\; v_2.\\ &\qquad (\mathcal{E}\; (\lambda n\; l\; v_3.\\ &\qquad\quad (k\; n\; l\; (\overrightarrow{\mathcal{J}}\; v_1\; v_2\; v_3)))\\ &\qquad\; n\; l\; \rho\; e_3))\\ &\quad\; n\; l\; \rho\; e_2))\\ &\;(n+1)\; l\;\; \rho\; e_1\end{aligned}\tag{26}$$

$$\begin{aligned}\mathcal{E}\; k\; n\; l\; \rho\; (\overleftarrow{\mathcal{J}}\; e_1\; e_2\; e_3) = \mathcal{E}\; &(\lambda n\; l\; v_1.\\ &\quad (\mathcal{E}\; (\lambda n\; l\; v_2.\\ &\qquad (\mathcal{E}\; (\lambda n\; l\; v_3.\\ &\qquad\quad (k\; n\; l\; (\overleftarrow{\mathcal{J}}\; v_1\; v_2\; v_3)))\\ &\qquad\; n\; l\; \rho\; e_3))\\ &\quad\; n\; l\; \rho\; e_2))\\ &\;(n+1)\; l\; \rho\; e_1\end{aligned}\tag{27}$$

Figure 9. Additions to the CPS evaluator for CHECKPOINTVLAD to support AD.

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 $\infty$. 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 7.4.

## 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 (Fig. 10). The implementation of PRIMOPS (29) calls the evaluator with no step limit and simply counts the number of steps to completion. The implementation of INTERRUPT (30) 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 (31) calls the evaluator with arguments from the saved capsule. Since the closure in the capsule does not use its argument, an arbitrary value $\perp$ is passed as that argument.

Note that the calls to $\mathcal{A}$ in $\overrightarrow{\mathcal{J}}$ (24), $\overleftarrow{\mathcal{J}}$ (25), PRIMOPS (29), INTERRUPT (30), and RESUME (31) are the only portals into the CPS evaluator. The only additional call to $\mathcal{A}$ is in the evaluator itself, clause (20) of $\mathcal{E}$. All of the portals restart the step count at zero. Except for the call in INTERRUPT (30), none of the portals call the evaluator with a step limit. In particular, RESUME (31) 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 can-

$$\textsc{primops}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x \tag{29}$$
$$\textsc{interrupt}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x \tag{30}$$
$$\textsc{resume}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot \tag{31}$$

Figure 10. Implementation of the general-purpose interruption and resumption interface using the CPS evaluator.

not be used to fully implement $\overset{\checkmark}{\mathcal{J}}$ in the host as depicted in Fig. 7. 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 such is what is invoked by the calls to $\mathcal{A}$ in $\overleftarrow{\mathcal{J}}$ (25) and INTERRUPT (30). As written in Fig. 7, the recursive calls to $\overset{\checkmark}{\mathcal{J}}$, namely steps (3) and (4), pass $(\lambda z.\textsc{resume}\ z)$ and $(\lambda x.\textsc{interrupt}\ f\ x\ \lfloor \frac{l}{2} \rfloor)$ 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 Fig. 7 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 Fig. 7 only uses it in the host. For INTERRUPT, the call in step (2) can use the host implementation (30) in Fig. 10 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 (31) in Fig. 10 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.

$$e ::= \textbf{interrupt}\ e_1\ e_2\ e_3 \mid \textbf{resume}\ e \tag{32}$$

We implement this functionality by augmenting the CPS evaluator with new clauses for $\mathcal{E}$ (Fig. 11), clause (35) for **interrupt** and clause (36) for **resume**. We discuss the implementation of these below. But we first address several other issues.

With appropriate implementations of **interrupt** and **resume** expressions in the target language, one can create target closures for the expressions $(\lambda z.\textsc{resume}\ z)$ and $(\lambda x.\textsc{interrupt}\ f\ x\ \lfloor \frac{l}{2} \rfloor)$, and use these to formulate a proper implementation of $\overset{\checkmark}{\mathcal{J}}$ in the host. We formulate a target closure to correspond to $(\lambda z.\textsc{resume}\ z)$ and denote this as $\mathcal{R}$. The definition is given in (34) in Fig. 11. Note that since $(\lambda z.\textsc{resume}\ z)$ does not contain any free variables, the closure created by $\mathcal{R}$ is constructed from the empty environment $\rho_0$. Thus there is a single constant $\mathcal{R}$. We similarly formulate a target closure to correspond to $(\lambda x.\textsc{interrupt}\ f\ x\ l)$ and denote this as $\mathcal{I}$. The definition is given in (33) in Fig. 11. Here, however, $(\lambda x.\textsc{interrupt}\ f\ x\ l)$ contains two free variables: $f$ and $l$. Thus the closure created by $\mathcal{I}$ contains a nonempty environment with values for these two variables. To provide these values, $\mathcal{I}$ is formulated as a function that takes these values as arguments.

$$
\begin{aligned}
\mathcal{I}\ f\ l &= \langle(\lambda x.(\textbf{interrupt}\ f\ x\ l)), \rho_0[f \mapsto f][l \mapsto l]\rangle && (33)\\
\mathcal{R} &= \langle(\lambda z.(\textbf{resume}\ z)), \rho_0\rangle && (34)\\
\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{interrupt}\ e_1\ e_2\ e_3) &= \mathcal{E}\ (\lambda n\ l\ v_1. && (35)\\
&\quad (\mathcal{E}\ (\lambda n\ l\ v_2.\\
&\quad\quad (\mathcal{E}\ (\lambda n\ l\ v_3.\\
&\quad\quad\quad \textbf{if}\ \ l = \infty\\
&\quad\quad\quad \textbf{then}\ (\mathcal{A}\ k\ 0\ v_3\ v_1\ v_2)\\
&\quad\quad\quad \textbf{else let}\ [\![k, f]\!] = (\mathcal{A}\ k\ 0\ l\ v_1\ v_2)\\
&\quad\quad\quad\quad \textbf{in}\ [\![k, (\mathcal{I}\ f\ (v_3 - l))]\!])\\
&\quad\quad\quad n\ l\ \rho\ e_3))\\
&\quad\quad n\ l\ \rho\ e_2))\\
&\quad (n+1)\ l\ \rho\ e_1\\
\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{resume}\ e) &= \mathcal{E}\ (\lambda n\ l\ [\![k', f]\!]. && (36)\\
&\quad (\mathcal{A}\ k'\ 0\ l\ f\ \bot)\\
&\quad (n+1)\ l\ \rho\ e
\end{aligned}
$$

Figure 11. Additions to the CPS evaluator for CHECKPOINTVLAD to expose the general-purpose interruption and resumption interface to the target.

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

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

Figure 12. Binary bisection checkpointing in the CPS evaluator. This is a proper implementation of the algorithm in Fig. 7 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 $f\ x\ \lfloor \frac{l}{2} \rfloor$) in step (4) is replaced with the target closure $(\mathcal{I}\ f\ \lfloor \frac{l}{2} \rfloor)$.

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 Fig. 7, replacing the host closure ($\lambda z.$RESUME $z$) in step (3) with the target closure $\mathcal{R}$ and the host closure ($\lambda x.$INTERRUPT $f\ x\ \lfloor \frac{l}{2} \rfloor$) in step (4) with the target closure $(\mathcal{I}\ f\ \lfloor \frac{l}{2} \rfloor)$. This new, proper, definition of $\overset{\checkmark}{\mathcal{J}}$ in the host is given in Fig. 12.

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 Fig. 12 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 Fig. 12. $\mathcal{I}$ is invoked two ways: step (4) of $\overset{\checkmark}{\mathcal{J}}$ in Fig. 12 and a way that we have not yet encountered, evaluation of nested **interrupt** expressions in the **else** branch of clause (35) in Fig. 11.

Consider all the ways that evaluations of $\mathcal{I}$ and $\mathcal{R}$ can be invoked in $\overset{\checkmark}{\mathcal{J}}$ in Fig. 12. $\overset{\checkmark}{\mathcal{J}}$ 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

$$(\mathcal{I}\,(\mathcal{I}\;\ldots\;(\mathcal{I}\,(\mathcal{I}\,f\,l_0)\,l_1)\;\ldots\;l_{i-1})\,l_i) \tag{37}$$

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 Fig. 12 always reduce $l$. The constructed target closures invoked in any other node in the binary checkpoint tree look like the following

$$(\mathcal{I}\,(\mathcal{I}\;\ldots\;(\mathcal{I}\,(\mathcal{I}\,\mathcal{R}\,l_0)\,l_1)\;\ldots\;l_{i-1})\,l_i) \tag{38}$$

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 Fig. 12. (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 (37) will result in a nested closure structure whose invocation will lead to nested invocations of **interrupt** expressions.

$$\begin{array}{l}
\langle(\lambda x.(\textbf{interrupt}\;f\;x\;l)),\\
\quad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\;f\;x\;l)),\\
\qquad\quad\rho_0[f\mapsto\langle\ldots\\
\qquad\qquad\quad(\lambda x.(\textbf{interrupt}\;f\;x\;l)),\\
\qquad\qquad\quad\rho_0[f\mapsto\langle(\lambda x.(\textbf{interrupt}\;f\;x\;l)),\\
\qquad\qquad\qquad\quad\rho_0[f\mapsto f]\\
\qquad\qquad\qquad\qquad[l\mapsto l_0]\rangle]\\
\qquad\qquad\qquad[l\mapsto l_1]\ldots\rangle]\\
\qquad\qquad[l\mapsto l_{i-1}]\rangle]\\
\quad\;[l\mapsto l_i]\rangle
\end{array} \tag{39}$$

A string of calls to $\mathcal{I}$ as in (38) will also result in a nested closure structure whose invocation will lead to nested invocations of **interrupt** expressions.

$$\begin{aligned}
&\langle(\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\quad\rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\qquad\qquad \rho_0[f \mapsto \langle\ldots\\
&\qquad\qquad\qquad (\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\qquad\qquad\qquad \rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\qquad\qquad\qquad\qquad \rho_0[f \mapsto \langle(\lambda z.(\textbf{resume}\ z)), \rho_0\rangle]\\
&\qquad\qquad\qquad\qquad\quad [l \mapsto l_0]\rangle]\\
&\qquad\qquad\qquad [l \mapsto l_1]\ldots\rangle]\\
&\qquad\qquad [l \mapsto l_{i-1}]\rangle]\\
&\quad [l \mapsto l_i]\rangle
\end{aligned} \tag{40}$$

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 (39) and $\langle(\lambda z.(\textbf{resume}\ z)), \rho_0\rangle$ in (40). 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

$$[\![k, f]\!] \tag{41}$$

where the closure $f$ contains only user code, *i.e.*, no **interrupt** or **resume** expressions. The $f$ in (41) is wrapped with calls to $\mathcal{I}$ to reintroduce the step limits $l_{i-1}, \ldots, l_1, l_0$.

$$[\![k, (\mathcal{I}\ \ldots\ (\mathcal{I}\ (\mathcal{I}\ f\ l_0)\ l_1)\ \ldots\ l_{i-1})]\!] \tag{42}$$

This will yield a capsule that looks like the following

$$\begin{aligned}
&[\![k, \langle(\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\quad \rho_0[f \mapsto \langle\ldots\\
&\qquad\qquad (\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\qquad\qquad \rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\ f\ x\ l)),\\
&\qquad\qquad\qquad \rho_0[f \mapsto f]\\
&\qquad\qquad\qquad\quad [l \mapsto l_0]\rangle]\\
&\qquad\quad [l \mapsto l_1]\ldots\rangle]\\
&\quad [l \mapsto l_{i-1}]\rangle]\!]
\end{aligned} \tag{43}$$

which will interrupt upon resumption. Each such interruption will peel off one **interrupt** expression. Note that since the closure $f$ in a capsule (41) contains only user code, it will not contain a **resume** expression. Further, since the wrapping process (43) only introduces **interrupt** expressions via calls to $\mathcal{I}$ (42), 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 (35) in Fig. 11. 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 (35) in Fig. 11. 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 (35).

Capsule resumption happens in one place, the call to $\mathcal{A}$ in clause (36) in Fig. 11 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 (31) in Fig. 10. 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 (31), 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 (39) or (40) wrapped around either a user function $f$ by (37) or a **resume** expression in $\mathcal{R}$ by (38), respectively. Such will always be invoked either by $\overleftarrow{\mathcal{J}}$ in the base case, step (0), or by INTERRUPT in step (2), of Fig. 12. The second is a preamble (43) wrapped around the closure of a capsule by the **else** branch in clause (35) of Fig. 11, *i.e.*, (42). Such will always be invoked during capsule resumption, *i.e.*, clause (36) of Fig. 11. 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 Fig. 12 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 Fig. 12 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 such. When we replace the CPS evaluator with a direct-style evaluator applied to CPS-converted target code, in Sections 3.11 and 3.12 below, 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.

$$e ::= \overset{\checkmark}{\mathcal{J}}\; e_1\; e_2\; e_3 \tag{44}$$

We implement this functionality by augmenting the CPS evaluator with a new clause (45) for $\mathcal{E}$ (Fig. 13).

$$
\begin{aligned}
\mathcal{E}\;k\;n\;l\;\rho\;(\overset{\checkmark}{\mathcal{J}}\;e_1\;e_2\;e_3) = \mathcal{E}\;&(\lambda n\;l\;v_1.\\
&\quad(\mathcal{E}\;(\lambda n\;l\;v_2.\\
&\qquad(\mathcal{E}\;(\lambda n\;l\;v_3.\\
&\qquad\quad(k\;n\;l\;(\overset{\checkmark}{\mathcal{J}}\;v_1\;v_2\;v_3)))\\
&\qquad\;\;n\;l\;\rho\;e_3))\\
&\quad\;\;n\;l\;\rho\;e_2))\\
&(n+1)\;l\;\rho\;e_1
\end{aligned}
\tag{45}
$$

Figure 13. Addition to the CPS evaluator for CHECKPOINTVLAD to support divide-and-conquer checkpointing.

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 Fig. 12 corresponds to Fig. 2(b). The start of the computation of $f$ in Fig. 12 corresponds to $u$ in Fig. 2(b). The computation state at $u$ is $x$ in Fig. 12. Collectively, the combination of $f$ and $x$ in Fig. 12 comprises a snapshot, the gold line in Fig. 2(b). The end of the computation of $f$ in Fig. 12 corresponds to $v$ in Fig. 2(b). The computation state at $v$ is $y$ in Fig. 12. Step (1) computes $\lfloor\frac{l}{2}\rfloor$ which corresponds to the split point $p$ in Fig. 2(b). Step (2) corresponds to the green line in Fig. 2(b), *i.e.*, running the primal without taping from the snapshot $f$ and $x$ at $u$ until the split point $p$ which is $\lfloor\frac{l}{2}\rfloor$. The capsule $z$ in Fig. 12 corresponds to the computation state at $p$ in Fig. 2(b). Brown and pink lines in Fig. 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 Fig. 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 Fig. 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 7.3), 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).

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 [13, Figs. 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 Fig. 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 Fig. 2 denote snapshots that are not capsules. Brown lines in Fig. 2 denote capsules that are not snapshots. Pink lines in Fig. 2 denote values that are both snapshots and capsules.

It is now easy to see that the recursive call tree of the algorithm in Fig. 12 is isomorphic to a binary checkpoint tree. The binary checkpoint tree on the left below corresponds to the call tree on the right produced by the algorithm in Fig. 12.

$$\begin{tikzcd}[arrows=dash]
& {[u,v)} \arrow[dl] \arrow[dr] & \\
{[u,p)} & & {[p,v)}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}[arrows=dash]
& z = (\textsc{interrupt}\ f\ x\ \lfloor\frac{l}{2}\rfloor) \arrow[dl] \arrow[dr] & \\
(z,\grave{x}) = (\overset{\checkmark}{\mathcal{J}}\ (\mathcal{I}\ f\ \lfloor\frac{l}{2}\rfloor)\ x\ \grave{z}) & & (y,\grave{z}) = (\overset{\checkmark}{\mathcal{J}}\ \mathcal{R}\ z\ \grave{y})
\end{tikzcd}$$

The above depicts just one level of the recursion. If one unrolls the above call tree to a depth of three one obtains the binary checkpoint tree depicted in Fig. 19.

## 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 [5].

We illustrate CPS conversion for the untyped lambda calculus (Fig. 14).

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \tag{46}$$

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 Fig. 14, (47) to (49), for each construct in (46). Clause (47) says that one converts a variable $x$ by calling the continuation $k$ with the value of that variable. Clause (48) says that one converts a lambda expression $(\lambda x.e)$ 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 (49) 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 (50) 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

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x \tag{47}$$
$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner) \tag{48}$$
$$\ulcorner (e_1\ e_2)|k \urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner \tag{49}$$
$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner \tag{50}$$

Figure 14. CPS conversion for the untyped lambda calculus.

language (Fig. 15). Clauses (51)–(60) correspond one-to-one to the CHECKPOINTVLAD constructs in (1), (10), and (44). 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 (61). Each of the clauses (51)–(60) 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 (Fig. 16). 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).

$$e ::= \lambda_3 n\ l\ x.e \mid \lambda_4 k\ n\ l\ x.e \mid e_1\ e_2\ e_3\ e_4 \mid e_1\ e_2\ e_3\ e_4\ e_5 \tag{62}$$

Second, we need to modify $\mathcal{E}$ to support these new constructs. We replace clause (2) with clauses (63) and (64) to update $\mathcal{A}$ and clauses (5) and (6) with clauses (65)–(68) to update $\mathcal{E}$. Third, we need to add support for **interrupt** and **resume** expressions, as is done with clauses (69) and (70). These are direct-style variants of clauses (35) and (36) 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 (35) and (36) are modified to use the converted form $\mathcal{A}_4$ of $\mathcal{A}$ (64) in (69) and (70). Similarly, the calls to continuations from (35) and (36) are modified to use the continuation form $\mathcal{A}_3$ of $\mathcal{A}$ (63) in (69) and (70). 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 (71) and (72). Note that unlike the corresponding (11) and (12), the calls to $\mathcal{A}_4$ here take target closures instead of host closures. Fifth, the general-purpose interruption and resumption interface, (29), (30), (33), and (34), must be migrated from the CPS evaluator to the direct-style evaluator as (73)–(76). In doing so, the calls to $\mathcal{A}_4$ in PRIMOPS and INTERRUPT are changed to use (64), the host continuations are modified to be target continuations in (73) and (74), and the lambda expressions in (75) and (76) are CPS converted.

$${}^{\ulcorner}_{\llcorner}c|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle k\ (n+1)\ l\ c\rangle\!\rangle_{k,n,l} \tag{51}$$

$${}^{\ulcorner}_{\llcorner}x|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l} \tag{52}$$

$${}^{\ulcorner}_{\llcorner}(\lambda x.e)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle k\ (n+1)\ l\ (\lambda_4 k\ n\ l\ x.{}^{\ulcorner}_{\llcorner}e|k,n,l{}^{\urcorner}_{\lrcorner})\rangle\!\rangle_{k,n,l} \tag{53}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(e_1\ e_2)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e_1|(\lambda_3 n\ l\ x_1.\\
\qquad\qquad {}^{\ulcorner}_{\llcorner}e_2|(\lambda_3 n\ l\ x_2.\\
\qquad\qquad\qquad (x_1\ k\ n\ l\ x_2)),\\
\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{54}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e_1|(\lambda_3 n\ l\ x_1.\\
\qquad\qquad (\mathbf{if}\ x_1\\
\qquad\qquad\ \ \mathbf{then}\ {}^{\ulcorner}_{\llcorner}e_2|k,n,l{}^{\urcorner}_{\lrcorner}\\
\qquad\qquad\ \ \mathbf{else}\ {}^{\ulcorner}_{\llcorner}e_3|k,n,l{}^{\urcorner}_{\lrcorner})),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{55}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(\diamond e)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e|(\lambda_3 n\ l\ x.\\
\qquad\qquad (k\ n\ l\ (\diamond x))),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{56}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(e_1 \bullet e_2)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e_1|(\lambda_3 n\ l\ x_1.\\
\qquad\qquad {}^{\ulcorner}_{\llcorner}e_2|(\lambda_3 n\ l\ x_2.\\
\qquad\qquad\qquad (k\ n\ l\ (x_1 \bullet x_2))),\\
\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{57}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e_1|(\lambda_3 n\ l\ x_1.\\
\qquad\qquad {}^{\ulcorner}_{\llcorner}e_2|(\lambda_3 n\ l\ x_2.\\
\qquad\qquad\qquad {}^{\ulcorner}_{\llcorner}e_3|(\lambda_3 n\ l\ x_3.\\
\qquad\qquad\qquad\qquad (k\ n\ l\ (\overrightarrow{\mathcal{J}}\ x_1\ x_2\ x_3))),\\
\qquad\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{58}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e_1|(\lambda_3 n\ l\ x_1.\\
\qquad\qquad {}^{\ulcorner}_{\llcorner}e_2|(\lambda_3 n\ l\ x_2.\\
\qquad\qquad\qquad {}^{\ulcorner}_{\llcorner}e_3|(\lambda_3 n\ l\ x_3.\\
\qquad\qquad\qquad\qquad (k\ n\ l\ (\overleftarrow{\mathcal{J}}\ x_1\ x_2\ x_3))),\\
\qquad\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{59}$$

$$\begin{array}{l}
{}^{\ulcorner}_{\llcorner}(\overset{\checkmark}{\mathcal{J}}\ e_1\ e_2\ e_3)|k,n,l{}^{\urcorner}_{\lrcorner} \leadsto \langle\!\langle {}^{\ulcorner}_{\llcorner}e_1|(\lambda_3 n\ l\ x_1.\\
\qquad\qquad {}^{\ulcorner}_{\llcorner}e_2|(\lambda_3 n\ l\ x_2.\\
\qquad\qquad\qquad {}^{\ulcorner}_{\llcorner}e_3|(\lambda_3 n\ l\ x_3.\\
\qquad\qquad\qquad\qquad (k\ n\ l\ (\overset{\checkmark}{\mathcal{J}}\ x_1\ x_2\ x_3))),\\
\qquad\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad\qquad\quad n,l{}^{\urcorner}_{\lrcorner}),\\
\qquad (n+1),l{}^{\urcorner}_{\lrcorner}\rangle\!\rangle_{k,n,l}
\end{array} \tag{60}$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \leadsto \mathbf{if}\ n=l\ \mathbf{then}\ [\![k,\lambda_4 k\ n\ l\ \_.e]\!]\ \mathbf{else}\ e \tag{61}$$

Figure 15. CPS conversion for the CHECKPOINTVLAD language that threads step counts and limits.

$$\mathcal{A}_3\ \langle(\lambda_3 n\ l\ x.e),\rho\rangle\ n'\ l'\ v = \mathcal{E}\ \rho[n\mapsto n'][l\mapsto l'][x\mapsto v]\ e \tag{63}$$

$$\mathcal{A}_4\ \langle(\lambda_4 k\ n\ l\ x.e),\rho\rangle\ k'\ n'\ l'\ v = \mathcal{E}\ \rho[k\mapsto k'][n\mapsto n'][l\mapsto l'][x\mapsto v]\ e \tag{64}$$

$$\mathcal{E}\ \rho\ (\lambda_3 n\ l\ x.e) = \langle(\lambda_3 n\ l\ x.e),\rho\rangle \tag{65}$$

$$\mathcal{E}\ \rho\ (\lambda_4 k\ n\ l\ x.e) = \langle(\lambda_4 k\ n\ l\ x.e),\rho\rangle \tag{66}$$

$$\mathcal{E}\ \rho\ (e_1\ e_2\ e_3\ e_4) = \mathcal{A}_3\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)\ (\mathcal{E}\ \rho\ e_4) \tag{67}$$

$$\mathcal{E}\ \rho\ (e_1\ e_2\ e_3\ e_4\ e_5) = \mathcal{A}_4\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)\ (\mathcal{E}\ \rho\ e_4)\ (\mathcal{E}\ \rho\ e_5) \tag{68}$$

$$\begin{aligned}\mathcal{E}\rho\ (\textbf{interrupt}\ e_1\ e_2\ e_3) = {} & \textbf{let}\ v_1 = (\mathcal{E}\ \rho\ e_1)\\ & \quad v_2 = (\mathcal{E}\ \rho\ e_2)\\ & \quad v_3 = (\mathcal{E}\ \rho\ e_3)\\ & \quad k = \rho\ \text{‘k’}\\ & \quad l = \rho\ \text{‘l’}\\ & \textbf{in if}\ \ l = \infty\\ & \quad \textbf{then}\ (\mathcal{A}_4\ v_1\ k\ 0\ v_3\ v_2)\\ & \quad \textbf{else let}\ [\![k, f]\!] = (\mathcal{A}_4\ v_1\ k\ 0\ l\ v_2)\\ & \qquad \textbf{in}\ [\![k, (\mathcal{I}\ f\ (v_3 - l))]\!]\end{aligned} \tag{69}$$

$$\begin{aligned}\mathcal{E}\ \rho\ (\textbf{resume}\ e) = {} & \textbf{let}\ [\![k', f]\!] = (\mathcal{E}\ \rho\ e)\\ & \quad l = \rho\ \text{‘l’}\\ & \textbf{in}\ (\mathcal{A}_4\ f\ k'\ 0\ l\ \bot)\end{aligned} \tag{70}$$

$$\begin{aligned}\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = {} & \textbf{let}\ (v_4 \rhd \acute{v}_5) = (\mathcal{A}_4\ v_1\ \langle(\lambda_3 n\ l\ v.v),\rho_0\rangle\ 0\ \infty\ (v_2 \rhd \acute{v}_3))\\ & \textbf{in}\ (v_4, \acute{v}_5)\end{aligned} \tag{71}$$

$$\begin{aligned}\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = {} & \textbf{let}\ (v_4 \lhd \grave{v}_5) = ((\mathcal{A}_4\ v_1\ \langle(\lambda_3 n\ l\ v.v),\rho_0\rangle\ 0\ \infty\ v_2) \lhd \grave{v}_3)\\ & \textbf{in}\ (v_4, \grave{v}_5)\end{aligned} \tag{72}$$

$$\textsc{primops}\ f\ x = \mathcal{A}_4\ f\ \langle(\lambda_3 n\ l\ v.n),\rho_0\rangle\ 0\ \infty\ x \tag{73}$$

$$\textsc{interrupt}\ f\ l\ n = \mathcal{A}_4\ f\ \langle(\lambda_3 n\ l\ v.v),\rho_0\rangle\ 0\ l\ x \tag{74}$$

$$\mathcal{I}\ f\ l = \langle(\lambda_4 k\ n\ l\ x.(\textbf{interrupt}\ f\ x\ l)),\rho_0[f\mapsto f][l\mapsto l]\rangle \tag{75}$$

$$\mathcal{R} = \langle(\lambda_4 k\ n\ l\ z.(\textbf{resume}\ z)),\rho_0\rangle \tag{76}$$

Figure 16. 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.

## 3.13 *Compiling Direct-Style Code to C*

One can compile target CHECKPOINTVLAD code, after CPS conversion, to C (Figs. 17 and 18). Modern implementations of C, like GCC, together with modern memory management technology, like the Boehm-Demers-Weiser garbage collector [8], allow the compilation process to be a straightforward mapping of each construct to a small 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 Figs. 17 and 18, $\mathcal{S}$ denotes such a mapping from CHECKPOINTVLAD expressions $e$ to C code fragments. Instead of environments $\rho$, $\mathcal{S}$ takes $\pi$, a mapping from variables to indices in `environment`, the run-time environment data structure. Here, $\pi\ x$ denotes the index of $x$, $\pi_i$ 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`,

$$\mathcal{S}\;\pi\;() = \texttt{null\_constant} \tag{77}$$
$$\mathcal{S}\;\pi\;\textbf{true} = \texttt{true\_constant} \tag{78}$$
$$\mathcal{S}\;\pi\;\textbf{false} = \texttt{false\_constant} \tag{79}$$
$$\mathcal{S}\;\pi\;(c_1, c_2) = \texttt{cons((}\mathcal{S}\;\pi\;c_1\texttt{), (}\mathcal{S}\;\pi\;c_1\texttt{))} \tag{80}$$
$$\mathcal{S}\;\pi\;n = n \tag{81}$$
$$\mathcal{S}\;\pi\;\texttt{`k'} = \texttt{continuation} \tag{82}$$
$$\mathcal{S}\;\pi\;\texttt{`n'} = \texttt{count} \tag{83}$$
$$\mathcal{S}\;\pi\;\texttt{`l'} = \texttt{limit} \tag{84}$$
$$\mathcal{S}\;\pi\;\texttt{`x'} = \texttt{argument} \tag{85}$$
$$\mathcal{S}\;\pi\;x = \texttt{as\_closure(target)->environment[}\pi\;x\texttt{]} \tag{86}$$
$$\mathcal{S}\;\pi\;(\lambda_3 n\;l\;x.e) = \texttt{(\{} \tag{87}$$

```
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 17. Compiler for the CHECKPOINTVLAD language when in CPS. Part I.

`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 Fig. 16, and the implementation of binary bisection checkpointing from Fig. 12 into C), allows arbitrary CHECKPOINTVLAD code 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.[8] All three accept exactly the same source language in its entirety and are able to run both examples discussed in Section 6. The first implementation is an interpreter based on the CPS evaluator (Figs. 8, 9, 11, and 13), where the evaluator, the operator overloading implementation of AD, the general-purpose interruption and resumption mechanism (Fig. 10), and the binary bisection checkpointing driver (Fig. 12) are implemented in

[8]The code for all three implementations will be released on `github` upon acceptance of this manuscript.

```
$\mathcal{S}\,\pi\,(\lambda_4 k\; n\; l\; x.e)$ = ({                                          (88)
                         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(($\mathcal{S}\,\pi\; e_1$),          (89)
                                              ($\mathcal{S}\,\pi\; e_2$),
                                              ($\mathcal{S}\,\pi\; e_3$),
                                              ($\mathcal{S}\,\pi\; e_4$))
$\mathcal{S}\,\pi\,(e_1\; e_2\; e_3\; e_4\; e_5)$ = converted_apply(($\mathcal{S}\,\pi\; e_1$),        (90)
                                              ($\mathcal{S}\,\pi\; e_2$),
                                              ($\mathcal{S}\,\pi\; e_3$),
                                              ($\mathcal{S}\,\pi\; e_4$),
                                              ($\mathcal{S}\,\pi\; e_5$))
$\mathcal{S}\,\pi\,(\textbf{if}\; e_1\; \textbf{then}\; e_2\; \textbf{else}\; e_3)$ = (!is_false(($\mathcal{S}\,\pi\; e_1$))?($\mathcal{S}\,\pi\; e_2$):($\mathcal{S}\,\pi\; e_3$))   (91)
$\mathcal{S}\,\pi\,(\diamond e)$ = ($\mathcal{N}\,\diamond$)(($\mathcal{S}\,\pi\; e$))                   (92)
$\mathcal{S}\,\pi\,(e_1 \bullet e_2)$ = ($\mathcal{N}\,\bullet$)(($\mathcal{S}\,\pi\; e_1$), ($\mathcal{S}\,\pi\; e_2$))       (93)
$\mathcal{S}\,\pi\,(\overrightarrow{\mathcal{J}}\; e_1\; e_2\; e_3)$ = ($\mathcal{N}\,\overrightarrow{\mathcal{J}}$)(($\mathcal{S}\,\pi\; e_1$), ($\mathcal{S}\,\pi\; e_2$), ($\mathcal{S}\,\pi\; e_3$))   (94)
$\mathcal{S}\,\pi\,(\overleftarrow{\mathcal{J}}\; e_1\; e_2\; e_3)$ = ($\mathcal{N}\,\overleftarrow{\mathcal{J}}$)(($\mathcal{S}\,\pi\; e_1$), ($\mathcal{S}\,\pi\; e_2$), ($\mathcal{S}\,\pi\; e_3$))   (95)
$\mathcal{S}\,\pi\,(\overset{\checkmark}{\mathcal{J}}\; e_1\; e_2\; e_3)$ = ($\mathcal{N}\,\overset{\checkmark}{\mathcal{J}}$)(($\mathcal{S}\,\pi\; e_1$), ($\mathcal{S}\,\pi\; e_2$), ($\mathcal{S}\,\pi\; e_3$))   (96)
```

Figure 18. Compiler for the CHECKPOINTVLAD language when in CPS. Part II.

SCHEME. The second implementation is a hybrid compiler/interpreter that translates the CHECKPOINTVLAD source program into CPS using CPS conversion (Fig. 15) and then interprets this with an interpreter based on the direct-style evaluator (Figs. 3, 4, and 16), where the compiler, the evaluator, the operator overloading implementation of AD, the general-purpose interruption and resumption mechanism (Fig. 16), and the binary bisection checkpointing driver (Fig. 12) are implemented in SCHEME. The third implementation is a compiler that translates the CHECKPOINTVLAD source program into CPS using CPS conversion (Fig. 15) and then compiles this to machine code via C using GCC, where the compiler (Figs. 17 and 18) is implemented in SCHEME, the evaluator is the underlying hardware, and the operator overloading implementation of AD, the general-purpose interruption and resumption mechanism (Fig. 16), and the binary bisection checkpointing driver (Fig. 7) are implemented in C. The first implementation was used to generate the results reported in [35] and presented at AD (2016). The techniques of Figs. 14 and 15 were presented at AD (2016). The third implementation was used to generate the results reported here.

## 4. Complexity

Fig. 19 illustrates the checkpoint tree that results from binary bisection checkpointing, our implementation of $\overset{\checkmark}{\mathcal{J}}$ (Fig. 12), with a recursion depth of three. The internal nodes 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 Fig. 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. We assume that $O(t) \geq O(w)$, *i.e.*, that the computation uses all storage.

The space complexity of classical reverse mode, including our implementation of $\overleftarrow{\mathcal{J}}$, is the sum of the space required for the primal and the space required for the tape. The space required for the primal is $O(w)$. The space required for the tape is $O(t)$. Thus the total space requirement is $O(w+t)$, which is $O(t)$ since $O(t) \geq O(w)$. The space overhead compared with computing just the primal is thus $O(t)$.

The time complexity of classical reverse mode, including our implementation of $\overleftarrow{\mathcal{J}}$, is the sum of the time required for the forward and reverse sweeps. The time required for the forward sweep is $O(t)$. The time required for the reverse sweep is also $O(t)$. Thus the total time requirement is $O(t)$. The time overhead compared with computing just the primal is thus $O(1)$.

The space complexity of binary bisection checkpointing, including our implementation of $\overset{\checkmark}{\mathcal{J}}$, is the sum of the space required for the primal, the snapshots, and the tape. When recursion is terminated when the step limit $l$ is below a fixed constant, the checkpoint tree has depth $O(\log t)$. The number of live snapshots is proportional to the depth of the checkpoint tree, $O(\log t)$. The size of each snapshot is proportional to the maximal live storage, $O(w)$. Thus the space required for the snapshots is $O(w \log t)$. The space required for the tape is proportional to the length of each stage, which is constant in our implementation. Thus the space required for the tape is $O(1)$. Thus the total space requirement is $O(w \log t)$ The space overhead compared with computing just the primal is thus $O(\log t)$.

The time complexity of binary bisection checkpointing, including our implementation of $\overset{\checkmark}{\mathcal{J}}$, is the sum of the time required to (re)compute the primal across all checkpoints and the time required to perform the forward and reverse sweeps across all stages. Each internal level in the checkpoint tree takes time proportional to the time required to run the entire primal, *i.e.*, $O(t)$. There are $O(\log t)$ levels. Thus the time required to (re)compute the primal across all checkpoints is $O(t \log t)$. The leaf level in the checkpoint tree takes time proportional to the time required to run the forward and reverse sweeps for the entire primal $f$ in classical reverse mode, *i.e.*, $O(t)$. Thus the total time requirement is $O(t \log t)$. The time overhead compared with computing just the primal is thus $O(\log t)$.

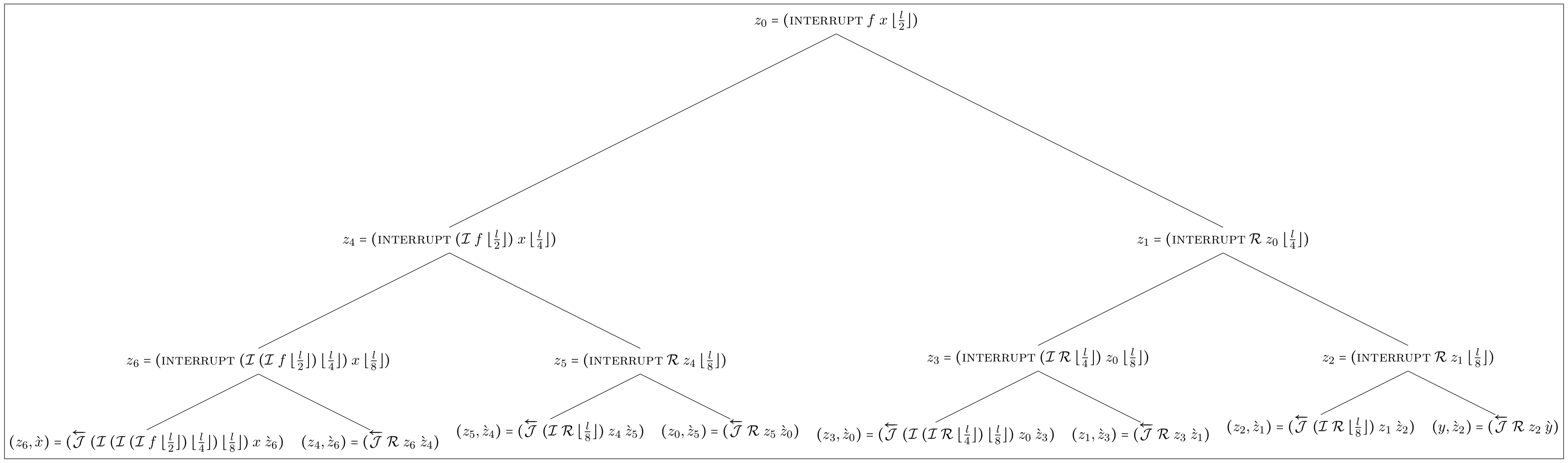

$z_0 = (\textsc{interrupt}\ f\ x\ \lfloor\frac{l}{2}\rfloor)$
$z_4 = (\textsc{interrupt}\ (\mathcal{I}\ f\ \lfloor\frac{l}{2}\rfloor)\ x\ \lfloor\frac{l}{4}\rfloor)$
$z_1 = (\textsc{interrupt}\ \mathcal{R}\ z_0\ \lfloor\frac{l}{4}\rfloor)$
$z_6 = (\textsc{interrupt}\ (\mathcal{I}\ (\mathcal{I}\ f\ \lfloor\frac{l}{2}\rfloor)\ \lfloor\frac{l}{4}\rfloor)\ x\ \lfloor\frac{l}{8}\rfloor)$
$z_5 = (\textsc{interrupt}\ \mathcal{R}\ z_4\ \lfloor\frac{l}{8}\rfloor)$
$z_3 = (\textsc{interrupt}\ (\mathcal{I}\ \mathcal{R}\ \lfloor\frac{l}{4}\rfloor)\ z_0\ \lfloor\frac{l}{8}\rfloor)$
$z_2 = (\textsc{interrupt}\ \mathcal{R}\ z_1\ \lfloor\frac{l}{8}\rfloor)$
$(z_6, \grave{x}) = (\overleftarrow{\mathcal{J}}\ (\mathcal{I}\ (\mathcal{I}\ (\mathcal{I}\ f\ \lfloor\frac{l}{2}\rfloor)\ \lfloor\frac{l}{4}\rfloor)\ \lfloor\frac{l}{8}\rfloor)\ x\ \grave{z}_6)$
$(z_4, \grave{z}_6) = (\overleftarrow{\mathcal{J}}\ \mathcal{R}\ z_6\ \grave{z}_4)$
$(z_5, \grave{z}_4) = (\overleftarrow{\mathcal{J}}\ (\mathcal{I}\ \mathcal{R}\ \lfloor\frac{l}{8}\rfloor)\ z_4\ \grave{z}_5)$
$(z_0, \grave{z}_5) = (\overleftarrow{\mathcal{J}}\ \mathcal{R}\ z_5\ \grave{z}_0)$
$(z_3, \grave{z}_0) = (\overleftarrow{\mathcal{J}}\ (\mathcal{I}\ (\mathcal{I}\ \mathcal{R}\ \lfloor\frac{l}{4}\rfloor)\ \lfloor\frac{l}{8}\rfloor)\ z_0\ \grave{z}_3)$
$(z_1, \grave{z}_3) = (\overleftarrow{\mathcal{J}}\ \mathcal{R}\ z_3\ \grave{z}_1)$
$(z_2, \grave{z}_1) = (\overleftarrow{\mathcal{J}}\ (\mathcal{I}\ \mathcal{R}\ \lfloor\frac{l}{8}\rfloor)\ z_1\ \grave{z}_2)$
$(y, \grave{z}_2) = (\overleftarrow{\mathcal{J}}\ \mathcal{R}\ z_2\ \grave{y})$

Figure 19. Binary bisection checkpoint tree of depth three. The checkpoint tree is traversed in depth-first right-to-left preorder.

## 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 [13, Fig. 4] as shown in Fig. 20. This supports the full functionality of that algorithm with the ability to select arbitrary execution points as split points. By selecting the choice of MID as either [13, equation (12)] or [13, equation (16)], one can select between bisection and binomial checkpointing. By selecting which of $d$, $t$, and $\alpha$ the user specifies, computing the others from the ones specified, together with $n$, using the methods described in [13] one can select the termination criterion to be either fixed space overhead, fixed time overhead, or logarithmic space and time overhead.[9] 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 Fig. 12 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 $\delta$ and a left-branch count $\tau$, 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 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 [13, equation (12)] or [13, equation (16)], one can select between bisection and binomial checkpointing. By selecting which of $d$, $t$, and $\alpha$ the user specifies, computing the others from the ones specified, together with $n$, using the methods described in [13] 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 [13], 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 $\alpha$ are mutually constrained by a single constraint.

$$n = \text{PRIMOPS}\ f\ x \tag{97}$$

$$\eta(d,t) = \binom{d+t}{t} \tag{98}$$

$$\alpha = \left\lceil \frac{n}{\eta(d,t)} \right\rceil \tag{99}$$

One can select any two and determine the third. Selecting $d$ and $\alpha$ to determine $t = O\left(\sqrt[d]{n}\right)$ yields the fixed space overhead termination criterion. Selecting $t$ and $\alpha$ to determine $d = O\left(\sqrt[t]{n}\right)$ yields the fixed time overhead termination criterion. Alterna-

[9]In Section 5 and Figs. 20 and 21 we use notation similar to that in [13] to facilitate understanding. Thus $n$ and $t$ here means something different then elsewhere in this manuscript.

```
$\textsc{treeverse}\ f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \beta\ \sigma\ \phi$ = if $\sigma > \beta$
                            then let $z = \textsc{interrupt}\ f\ x\ (\sigma - \beta)$
                                in $\textsc{first}\ \mathcal{R}\ z\ \grave{y}\ \alpha\ (\delta - 1)\ \tau\ \beta\ \sigma\ \phi$
                            else $\textsc{first}\ f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \beta\ \sigma\ \phi$
$\textsc{first}\ f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \beta\ \sigma\ \phi$ = if $\phi - \sigma > \alpha \wedge \delta \neq 0 \wedge \tau \neq 0$
                            then let $\kappa = \textsc{mid}\ \delta\ \tau\ \sigma\ \phi$
                                    $(y, \grave{z}) = \textsc{treeverse}\ f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \sigma\ \kappa\ \phi$
                                in $\textsc{rest}\ f\ x\ \grave{z}\ \alpha\ y\ \delta\ (\tau - 1)\ \beta\ \sigma\ \kappa$
                            else $\overleftarrow{\mathcal{J}}\ (\mathcal{I}\ f\ (\phi - \sigma))\ x\ \grave{y}$
$\textsc{rest}\ f\ x\ \grave{y}\ \alpha\ y\ \delta\ \tau\ \beta\ \sigma\ \phi$ = if $\phi - \sigma > \alpha \wedge \delta \neq 0 \wedge \tau \neq 0$
                            then let $\kappa = \textsc{mid}\ \delta\ \tau\ \sigma\ \phi$
                                    $(\_, \grave{z}) = \textsc{treeverse}\ f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \sigma\ \kappa\ \phi$
                                in $\textsc{rest}\ f\ x\ \grave{z}\ \alpha\ y\ \delta\ (\tau - 1)\ \beta\ \sigma\ \kappa$
                            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 $n = \textsc{primops}\ f\ x$
                                $\textsc{pick}\ \alpha\ d\ t$
                            in $\textsc{treeverse}\ f\ x\ \grave{y}\ \alpha\ d\ t\ 0\ 0\ n$
```

Figure 20. Implementation of TREEVERSE from [13, Fig. 4] using the general-purpose interruption and resumption interface, written in a functional style with no mutation. The variables $\delta$, $\tau$, $\beta$, $\sigma$, $\phi$, $n$, $d$, and $t$ have the same meaning as in [13]. The variables $f$, $x$, $\grave{x}$, $y$, $\grave{y}$, $z$, and $\grave{z}$ have the same meaning as earlier in this manuscript. The variable $\alpha$ denotes an upper bound on the number of evaluation steps for a leaf node. Different termination criteria allow the user to specify some of $\alpha$, $d$, and $t$ and compute the remainder as a function of the ones specified, together with $n$.

```
$\textsc{binary}\ f\ x\ \grave{y}\ \alpha\ \delta\ \tau\ \phi$ = if $\phi \leq \alpha \vee \delta = 0 \vee \tau = 0$
                        then $\overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$
                        else let $\kappa = \textsc{mid}\ \delta\ \tau\ 0\ \phi$
                                $z = \textsc{interrupt}\ f\ x\ \kappa$
                                $(y, \grave{z}) = \textsc{binary}\ \mathcal{R}\ z\ \grave{y}\ (\delta - 1)\ \tau\ (\phi - \kappa)$
                                $(z, \grave{x}) = \textsc{binary}\ (\mathcal{I}\ f\ \kappa)\ x\ \grave{z}\ \delta\ (\tau - 1)\ \kappa$
                            in $(y, \grave{x})$

            $\overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$ = let $n = \textsc{primops}\ f\ x$
                            $\textsc{pick}\ \alpha\ d\ t$
                        in $\textsc{binary}\ f\ x\ \grave{y}\ \alpha\ d\ t\ n$
```

Figure 21. Implementation of binary checkpointing using the general-purpose interruption and resumption interface in a fashion that supports all of the functionality of TREEVERSE from [13, Fig. 4]. The variables $\delta$, $\tau$, $\phi$, $n$, $d$, and $t$ have the same meaning as in [13]. The variables $f$, $x$, $\grave{x}$, $y$, $\grave{y}$, $z$, and $\grave{z}$ have the same meaning as earlier in this manuscript. The variable $\alpha$ denotes an upper bound on the number of evaluation steps for a leaf node. Different termination criteria allow the user to specify some of $\alpha$, $d$, and $t$ and compute the remainder as a function of the ones specified, together with $n$.

tively, one can further constrain $d = t$. With this, selecting $\alpha$ to determine $d$ and $t$ yields the logarithmic space and time overhead termination criterion.

## 6. Examples

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 are arbitrary execution points. The example in Listing 1 illustrates a situation where placing split points only at loop iteration boundaries can fail to 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 Listing 2.

For comparison, we reformulate this FORTRAN example in CHECKPOINTVLAD (Listing 2) 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 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 $w = O(1)$, since there are no arrays; the amount of storage taken by the primal is constant. The time required for the primal for this example should be $t = O(n)$, since there are $n$ 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 derivatives on this particular example:

| | space | time |
|---|---|---|
| primal | $O(1)$ | $O(n)$ |
| TAPENADE no checkpointing | $O(n)$ | $O(n)$ |
| TAPENADE divide-and-conquer checkpointing | $O(1)$ | $O\left(n\sqrt[d]{n}\right)$ |
| CHECKPOINTVLAD no checkpointing | $O(n)$ | $O(n)$ |
| CHECKPOINTVLAD divide-and-conquer checkpointing | $O(\log n)$ | $O(n \log n)$ |

The efficacy of our method can be seen in the plots (Fig. 22) of the observed space and time usage of the above two FORTRAN variants and the above two CHECKPOINTVLAD variants with varying $n$. We observe that TAPENADE space and time usage grows with $n$ for all cases. CHECKPOINTVLAD space and time usage grows with $n$ with $\overleftarrow{\mathcal{J}}$. CHECKPOINTVLAD space usage is constant with $\overset{\checkmark}{\mathcal{J}}$. CHECKPOINTVLAD time usage grows with $n$ with $\overset{\checkmark}{\mathcal{J}}$. Only the first three data points are shown for the CHECKPOINTVLAD

Listing 2 A rendering of the example from Listing 1 in CHECKPOINTVLAD. Space and time overhead of two variants of this example when run under CHECKPOINTVLAD are presented in Fig. 22. The variant for divide-and-conquer checkpointing is shown. The variant with no checkpointing replaces `checkpoint-*j` with `*j`.

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

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

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

(define (f n x)
 (let outer ((i 1) (y x))
  (if (> i n)
      y
      (outer
       (+ i 1)
       (let ((m (expt
                 2
                 (- (ilog2 n)
                    (ilog2
                     (+ 1 (modulo (* (* (floor (expt 3 x)) i)
                                     1007)
                                  n)))))))
        (let inner ((j 1) (y y))
         (if (> j m)
             y
             (inner (+ j 1) (sqrt (* y y))))))))))

(let* ((n (read-real))
       (x (read-real))
       (y-grave (read-real))
       (result (checkpoint-*j (lambda (x) (f n x)) x y-grave)))
 (cons (write-real (car result)) (write-real (cdr result))))
```

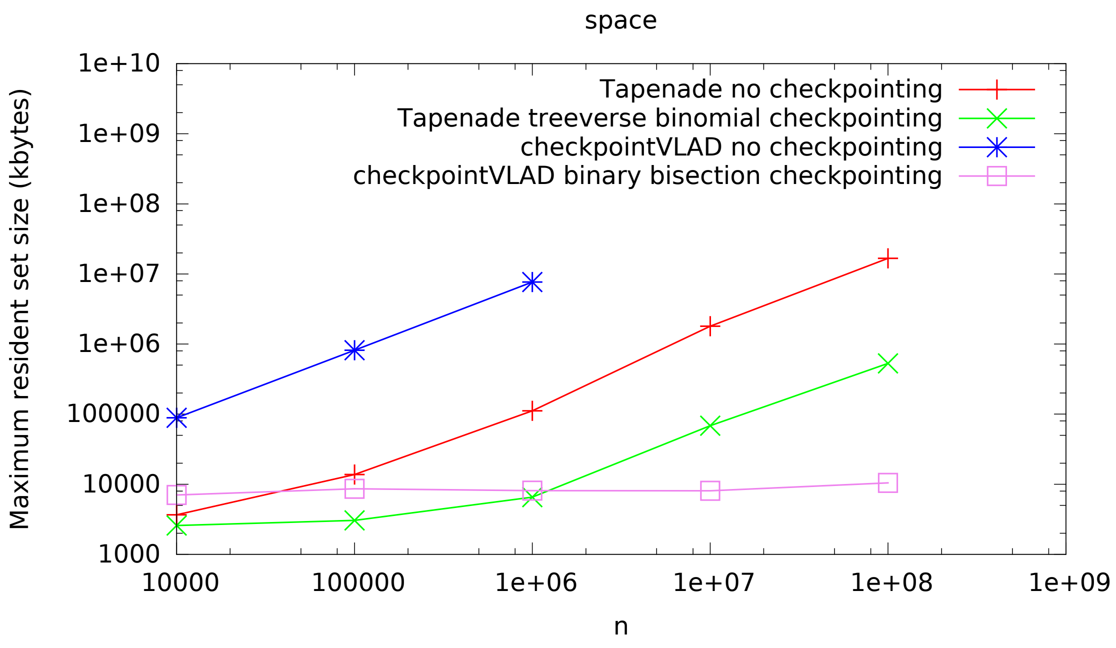
variant without checkpointing as the tape needed for this example exceeds our available RAM with large $n$.

The crucial aspect of this example is that we observe sublinear space usage overhead with divide-and-conquer checkpointing in CHECKPOINTVLAD but not with such in TAPENADE.[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 does not exceed the asymptotic space complexity of the whole primal computation. In this case, the sublinear space overhead guarantee would only hold if the asymptotic time complexity of the loop body is $O(1)$. Since the asymptotic time complexity of the loop body is $O(n)$, the requisite tape size grows with $O(n)$ even though the number of snapshots, and the size of those snapshots, is bounded by a constant.

The purpose of this example is simply to illustrate our central claim:

> Situations arise where placing split points only at loop iteration boundaries can fail to 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-

[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. The observed overhead appears to be constant. We are not sure why. Perhaps, the memory usage is so low, under about 10MB, that the fixed memory required by the code, the libraries, the stack, and memory management swamps the memory usage for the snapshots and tape. Moreover, the memory hierarchy of modern computers, *i.e.*, caches, translate lookaside buffers, virtual memory, operating-system overhead, and the algorithms employed by memory management obscure time measurement. Furthermore, a logarithmic factor for both space and time is difficult to observe in log-log plots. But nothing turns on this. Asymptotic complexity is a model that holds in the limit; observations short of that limit often fail to fit the model. In our case, we are doing better than the model prediction, not worse. Our next example, below, shows that our implementation of CHECKPOINTVLAD does indeed exhibit the theoretical logarithmic space and time usage overhead.



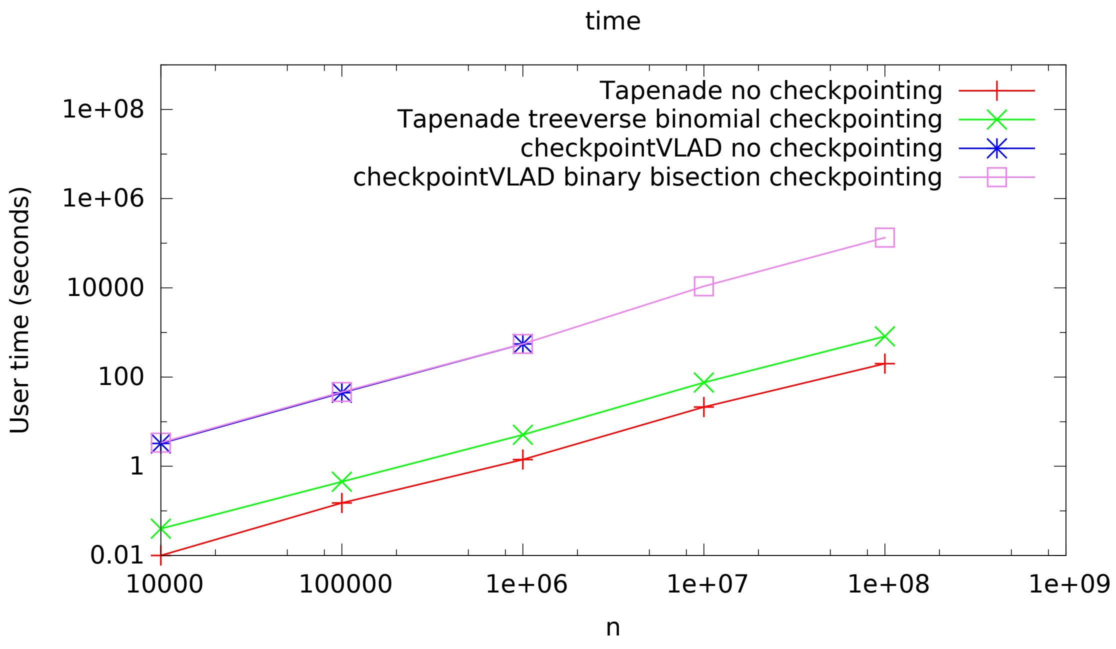


Figure 22. Space and time usage of reverse-mode AD with and without divide-and-conquer checkpointing for the example in Listings 1 and 2. Space and time usage was measured with `/usr/bin/time --verbose`.

conquer checkpointing.

This example illustrates that divide-and-conquer checkpointing in Tapenade, including binomial checkpointing, can fail to yield sublinear space overhead in situations where checkpointVLAD can exhibit such.

Our contribution is the method discussed in Section 3 for building language implementations, both interpreted and compiled, that support divide-and-conquer checkpointing where placement of split points at arbitrary execution points will yield the desired sublinear space and time overhead. We have implemented our method, both as an interpreter, as a hybrid compiler/interpreter, and as a compiler, to illustrate our contribution and provide concrete documentation to others who wish to build upon our work. We do not view our implementation artifacts themselves as contributions. We make no claim that our implementations are particularly useful or efficient. They may exhibit poor constant factors and otherwise be unusable. Thus the comparison of absolute space and time usage between Tapenade and checkpointVLAD is unwarranted. The plots in Fig. 22 serve solely to illustrate our central claim and the fact that our method succeeds in achieving our objective: sublinear space overhead of divide-and-conquer checkpointing with placement of split points at arbitrary execution points. We leave it to others to build upon our work and incorporate our method into a useful and efficient implementation.

To illustrate that our method has the potential to scale to larger programs, we run a slightly larger example, computing the gradient of the determinant of a matrix, a variation of an example from [13]. Here, the determinant is computed by Gaussian elimination without pivoting. To reduce the output to a single number that can be checked for correctness, we compute the determinant of the gradient of the determinant of the identity matrix. We do this for $n \times n$ matrices of varying $n$. Fortran and checkpointVLAD implementations of this example are shown in Listings 3 and 4. Again, we run the variants with Tapenade and checkpointVLAD the same two ways as the previous example.

For this example, $n$ is the number of rows or columns in the matrix. Using the notation from Section 4, the maximal space usage of the primal for this example should be $w = O(n^2)$. The time required by the primal for this example should be $t = O(n^3)$. However, checkpointVLAD does not provide arrays with constant time access and update; such is emulated with nested lists and has $O(n)$ access and update. Thus the time required by the primal for this example with checkpointVLAD should be $t = O(n^4)$. The analysis in Sections 4 and 5 predicts the following asymptotic space and time complexity of the Tapenade and checkpointVLAD variants that compute derivatives on this particular example:

| | space | time |
|---|---|---|
| Fortran primal | $O(n^2)$ | $O(n^3)$ |
| Tapenade no checkpointing | $O(n^3)$ | $O(n^3)$ |
| Tapenade divide-and-conquer checkpointing | $O(n^2)$ | $O\left(n^3 \sqrt[d]{n}\right)$ |
| checkpointVLAD primal | $O(n^2)$ | $O(n^4)$ |
| checkpointVLAD no checkpointing | $O(n^3)$ | $O(n^4)$ |
| checkpointVLAD divide-and-conquer checkpointing | $O(n^2 \log n)$ | $O(n^4 \log n)$ |

In this case, the fixed space overhead guarantee of Tapenade should hold because the asymptotic time complexity of the body of the outer loop is $O(n^2)$ which is the same as the asymptotic time complexity of the whole primal. Nominally, the tape required by checkpointVLAD without checkpointing should be $O(n^4)$. But even though that is the time complexity, the tape length is the number of arithmetic operations, which is only $O(n^3)$.

Observed space and time usage for varying $n$ of the Fortran/Tapenade variants are shown in Fig. 23. Those for the checkpointVLAD variants are shown in Fig. 24. We overlay fits of the expected complexity classes obtained by linear regression on all but the space and time usage of the Fortran/Tapenade variant without checkpointing because these fits are poor for unknown reasons. Such is not our concern here. The other fits seem quite good. This suggests that, on this example, divide-and-conquer checkpointing in both Tapenade and checkpointVLAD exhibit the expected sublinear space and time overhead.

Listing 3 FORTRAN determinant example. This example is rendered in CHECKPOINTVLAD in Listing 4. Space and time overhead of two variants of this example when run under TAPENADE are presented in Fig. 23. 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.

```
      subroutine ident(n, a)
      double precision a(n, n)
      do i = 1, n
         do j = 1, n
            a(i, j) = 0.0d0
            if (i.eq.j) a(i, j) = 1.0d0
         end do
      end do
      end

      subroutine det(n, a, d)
      include 'determinant.inc'
      double precision a(n, n), b(nn, nn), c, d, e
      do i = 1, n
         do j = 1, n
            b(i, j) = a(i, j)
         end do
      end do
      d = 1.0d0
c$ad binomial-ckp n+1 30 1
      do i = 1, n
         c = b(i, i)
         d = d*c
         do j = i, n
            b(i, j) = b(i, j)/c
         end do
         do j = i+1, n
            e = b(j, i)
            do k = i+1, n
               b(j, k) = b(j, k)-e*b(i, k)
            end do
         end do
      end do
      end

      program main
      include 'determinant.inc'
      double precision a(nn, nn), ab(nn, nn), d, db
      read *, n
      call ident(n, a)
      call det(n, a, d)
      db = 1.0d0
      call det_b(n, a, ab, d, db)
      call det(n, ab, d)
      print *, d
      end
```

## 7. Discussion

The techniques described above are relevant to current research trends and can be made efficient with currently available implementation technology.

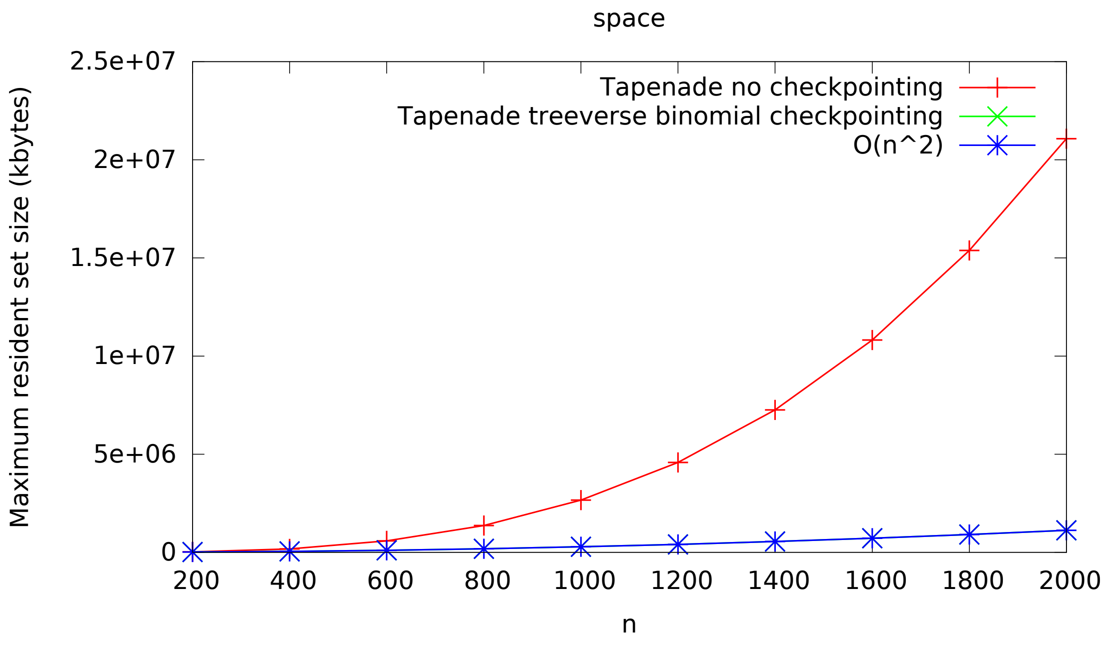
### 7.1 *Deep Learning*

Simple deep or recurrent neural networks are systems of the basic form $y(t+1) = f(y(t), w)$ where $w$ is to be found so as to optimize some functional $E$ of the mapping $y(0) \mapsto y(T)$. This is typically done using a gradient method, with the gradient $\nabla_w E$ calculated using reverse-mode AD [25]. A variety of methods have been developed to allow this optimization to be successful, under the general rubric of "Deep Learning" [12, 22, 31]. The storage overhead of doing this has, for both very deep networks and recurrent networks that run for a considerable number of time steps, come to severely

Listing 4 A rendering of the example from Listing 3 in CHECKPOINTVLAD. Space and time overhead of two variants of this example when run under CHECKPOINTVLAD are presented in Fig. 24. The variant for divide-and-conquer checkpointing is shown. The variant with no checkpointing replaces `checkpoint-*j` with `*j`.

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

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

(define (matrix-rows a)
 (if (null? a) 0 (+ (matrix-rows (cdr a)) 1)))

(define (list-ref l i)
 (if (zero? i) (car l) (list-ref (cdr l) (- i 1))))

(define (matrix-ref a i j) (list-ref (list-ref a i) j))

(define (list-set l i x)
 (if (zero? i)
     (cons x (cdr l))
     (cons (car l) (list-set (cdr l) (- i 1) x))))

(define (matrix-set a i j x)
 (list-set a i (list-set (list-ref a i) j x)))

(define (map-n f n)
 (if (zero? n) '() (cons (f (- n 1)) (map-n f (- n 1)))))

(define (identity-matrix n)
 (map-n (lambda (i) (map-n (lambda (j) (if (= i j) 1 0)) n)) n))

(define (determinant a)
 (let ((n (matrix-rows a)))
  (let loop ((i 0) (b a) (d 1))
   (if (= i n)
       d
       (let* ((c (matrix-ref b i i))
              (b (let loop ((j i) (b b))
                  (if (= j n)
                      b
                      (loop (+ j 1)
                            (matrix-set
                             b i j (/ (matrix-ref b i j) c)))))))
        (loop
         (+ i 1)
         (let loop ((j (+ i 1)) (b b))
          (if (= j n)
              b
              (loop
               (+ j 1)
               (let ((e (matrix-ref b j i)))
                (let loop ((k (+ i 1)) (b b))
                 (if (= k n)
                     b
                     (loop (+ k 1)
                           (matrix-set
                            b j k
                            (- (matrix-ref b j k)
                               (* e (matrix-ref b i k)))))))))))
         (* d c)))))))

(write-real
 (determinant
  (cdr
   (checkpoint-*j determinant (identity-matrix (read-real)) 1))))
```

tax the limits of existing platforms. In response to this, variants of divide-and-conquer checkpointing in reverse-mode AD have been rediscovered and deployed by that community [9, 16]. These implementations are far from automatic, and depend on compile-time analysis of the static primal flow graphs. For this reason, the work on divide-and-conquer checkpointing in reverse mode in the AD community may have important applications



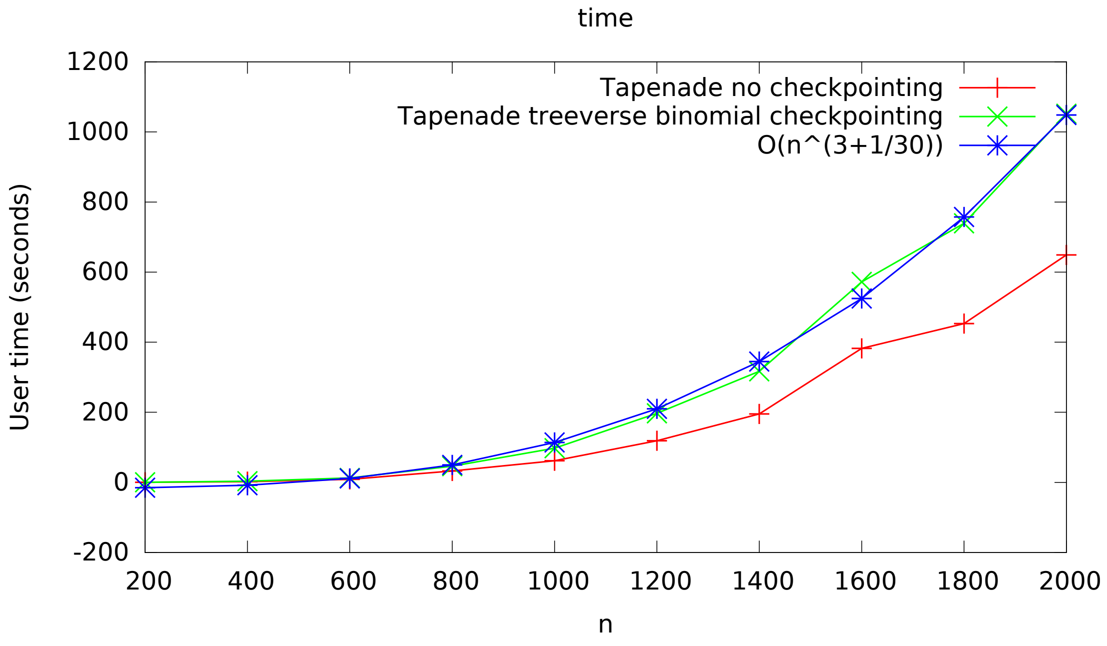
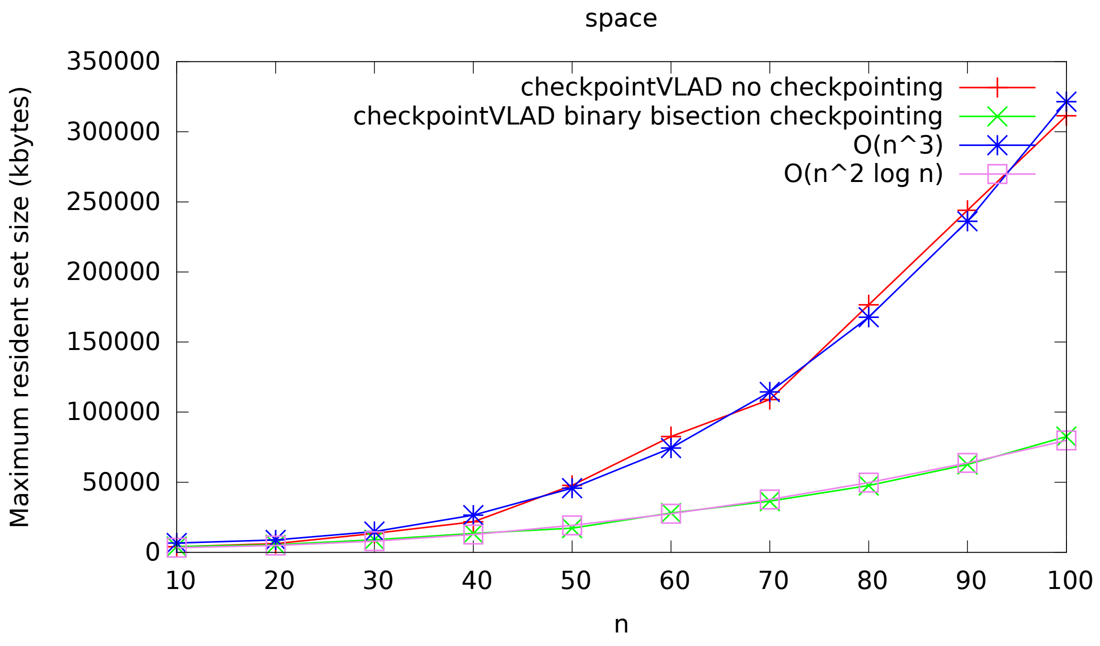


Figure 23. Space and time usage of reverse-mode AD with and without divide-and-conquer checkpointing for the example in Listing 3. Space and time usage was measured with `/usr/bin/time --verbose`.

in deep learning—and the deep learning community may have new ideas of interest to the AD community as well [6, 23].

## 7.2 *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



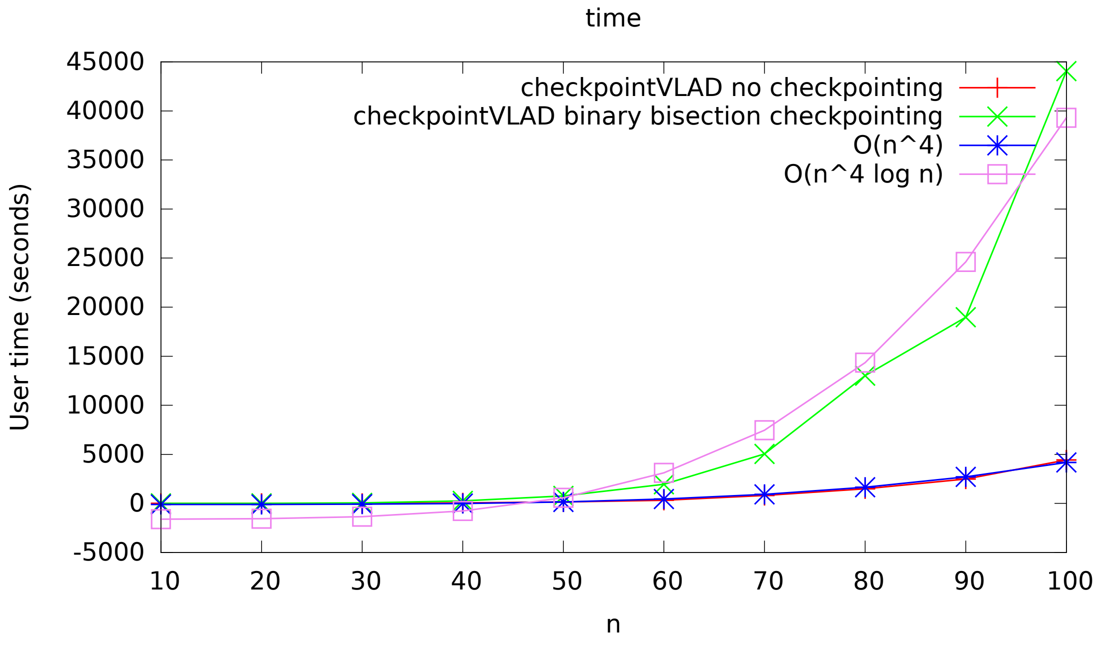


Figure 24. Space and time usage of reverse-mode AD with and without divide-and-conquer checkpointing for the example in Listings 4. Space and time usage was measured with `/usr/bin/time --verbose`.

used to provide an overloaded implementation of divide-and-conquer checkpointing in a fashion that was largely transparent to the user [13]. However, the Associate Editor of this journal has reported that

> It was found to be rather slow and not further pursued by the author.

Indeed, [13] reports:

> For example, the calculation of a $10 \times 10$ determinant using Legendre's rule involves

> 10! = 3,628,800 multiplications and additions or subtractions. Since the determinant was computed using a recursive function call, many assignments and so-called death notices had to be recorded. These overhead operations brought the total length of the tape to almost $T = 814$ megabytes, corresponding to nearly a million computational steps of one kilobyte size. As predicted by the theory this problem could be solved for the combination $d = t = 12$. The computing time was extensive, because all forward sweeps were performed with recording for programming simplicity. An efficient version is currently being developed.

and the last paragraph of [13] 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. As shown in Section 6, not only can we compute the derivative of the determinant of a $10 \times 10$ matrix with a total of 4048 kilobytes in 0.38 seconds, we can do so for a $100 \times 100$ matrix with a total of 82560 kilobytes in 44056.15 seconds. Even with this, our implementation 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$\nabla$) 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 to add such. 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 [5], for functional languages like SML, already generate target code in CPS, so it seems feasible to adapt such 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 [44], for languages like SML, that achieve very low overhead for automatic storage management.

[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 [4, 20].

## 7.3 *Advantages of Functional Languages for Interruption and Resumption*

The reason that CHECKPOINTVLAD does not support linear-time array update is that functional languages do not support assignment statements. Thus array updates involve copying. Functional languages simplify interruption and resumption, allowing such 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. The ability to mutate structure with assignment statements would foil all that.

## 7.4 *Nesting of AD Operators*

It has been argued that the ability to nest AD operators is important in many practical domains [2, 3, 10, 23, 27–29]. Supporting nested use of AD operators involves many subtle issues [24, 32, 33]. CHECKPOINTVLAD addresses these issues and fully supports nested use of AD operators. One can write programs of the form

$$\alpha\ (\lambda x.(\ldots(\beta\ f\ \ldots)))\ \ldots \tag{100}$$

where each of $\alpha$ and $\beta$ 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 (26), (27), (45), and (58)–(60). What this means is that if $\alpha$ in (100) were $\overset{\checkmark}{\mathcal{J}}$, a split point for $\alpha$ 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 $\epsilon$ tag as described in [32]. The $\epsilon$ 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 $\epsilon$ tag. This requires that the entries to and exits from AD operator invocations obey last-in-first-out sequencing. If $\alpha$ were $\overset{\checkmark}{\mathcal{J}}$, and the computation of $f$ were interrupted, then situations could arise where the last-in-first-out sequencing of $\beta$ 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 accumulations 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 $\alpha$ and $\beta$ were $\overset{\checkmark}{\mathcal{J}}$, and $\overset{\checkmark}{\mathcal{J}}$ was not atomic, situations could arise where $f$ could interrupt for $\alpha$ instead of $\beta$. 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 $\alpha$ instead of $\beta$. This situation resembles a situation that arises with probabilistic programming languages. Probabilistic programming languages provide a sampling operation, a mechanism for sampling from a distribution, and an expectation operator, a mechanism for aggregating samples into the expectation of a distribution. Nesting such must obey a last-in-first-out sequencing: a sampling operation must be performed relative to the immediate surrounding expectation operator invocation, not to one outside that at a higher scope. 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 favorable storage requirements of reverse mode with divide-and-conquer checkpointing, guaranteeing sublinear space and time overhead.

## Disclosure

This manuscript is an expanded version of the extended abstract of a presentation at AD (2016), which is available as a technical report [35]. 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/.

## Funding

This work was supported, in part, by the US National Science Foundation under Grant 1522954-IIS and by Science Foundation Ireland under Grant SFI 09/IN.1/I2637. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

## References

[1] P. Achten, J. Van Groningen, and R. Plasmeijer, *High level specification of I/O in functional languages*, in *Functional Programming, Glasgow 1992*, Springer, 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*, Available at https://arxiv.org/abs/1606.04474, NIPS, 2016.
[4] A.W. Appel, *SSA is functional programming*, ACM SIGPLAN Notices 33 (1998), pp. 17–20.
[5] A.W. Appel, *Compiling with continuations*, Cambridge University Press, 2006.
[6] A.G. Baydin, B.A. Pearlmutter, and J.M. Siskind, *Tricks from deep learning* (2016), Available at https://arxiv.org/abs/1611.03777.
[7] 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.
[8] H. Boehm, A. Demers, and M. Weiser, *A garbage collector for C and C++*, Available at http://www.hboehm.info/gc/.
[9] 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.
[10] B. Christianson, *Reverse accumulation of functions containing gradients*, Tech. Rep. 278, University of Hertfordshire Numerical Optimisation Centre, 1993, Available at http://hdl.handle.net/2299/4337, presented at the Theory Institute Argonne National Laboratory Illinois, Procs. of the Theory Institute on Combinatorial Challenges in Automatic Differentiation.
[11] 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, Springer, Heidelberg, 2006, pp. 566–573.
[12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, MIT Press, 2016, Available at http://www.deeplearningbook.org.
[13] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software 1 (1992), pp. 35–54.
[14] 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, this version of the manual is superceded by http://www.math.tu-dresden.de/~adol-c/adolc110.ps.
[15] A. Griewank and A. Walther, *Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation*, ACM Transactions on Mathematical Software 26 (2000), pp. 19–45, also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.
[16] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-Efficient Backpropagation Through Time*, in *Neural Information Processing Systems*, Available at https://arxiv.org/abs/1606.03401, NIPS, 2016.
[17] L. Hascoët and V. Pascual, *TAPENADE 2.1 user's guide*, Rapport technique 300, INRIA, 2004.

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

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

[20] 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 30 (1995), pp. 13–22.

[21] 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, Springer, Heidelberg, 2006, pp. 566–573.

[22] Y. LeCun, Y. Bengio, and G.E. Hinton, *Deep learning*, Nature 521 (2015), pp. 436–444.

[23] D. Maclaurin, D. Duvenaud, and R.P. Adams, *Gradient-based hyperparameter optimization through reversible learning* (2015), Available at https://arxiv.org/abs/1502.03492.

[24] O. Manzyuk, B.A. Pearlmutter, A.A. Radul, D.R. Rush, and J.M. Siskind, *Confusion of tagged perturbations in forward automatic differentiation of higher-order functions* (2012), Available at https://arxiv.org/abs/1211.4892.

[25] B.A. Pearlmutter, *Gradient calculation for dynamic recurrent neural networks: a survey*, IEEE Transactions on Neural Networks 6 (1995), pp. 1212–1228.

[26] B.A. Pearlmutter and J.M. Siskind, *Reverse-mode AD in a functional framework: Lambda the ultimate back propagator*, ACM Transactions on Programming Languages and Systems 30 (2008), pp. 7:1–7:36.

[27] B.A. Pearlmutter and J.M. Siskind, *Using programming language theory to make automatic differentiation sound and efficient*, in *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, and J. Utke, eds., Lecture Notes in Computational Science and Engineering, Vol. 64, Springer, Berlin, 2008, pp. 79–90.

[28] A. Radul, B.A. Pearlmutter, and J.M. Siskind, *AD* in *Fortran: Implementation via prepreprocessor*, in *Recent Advances in Algorithmic Differentiation*, S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, eds., Lecture Notes in Computational Science and Engineering, Vol. 87, Springer, Berlin, 2012, pp. 273–284, Available at https://arxiv.org/abs/1203.1450.

[29] A. Radul, B.A. Pearlmutter, and J.M. Siskind, *AD in Fortran, Part 1: Design* (2012), Available at https://arxiv.org/abs/1203.1448.

[30] J.C. Reynolds, *The discoveries of continuations*, Lisp and Symbolic Computation 6 (1993), pp. 233–247.

[31] J. Schmidhuber, *Deep learning in neural networks: An overview*, Neural Networks 61 (2015), pp. 85–117.

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

[33] J.M. Siskind and B.A. Pearlmutter, *Putting the automatic back into AD: Part I, What's wrong*, Tech. Rep. TR-ECE-08-02, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, 2008, Available at http://docs.lib.purdue.edu/ecetr/368.

[34] J.M. Siskind and B.A. Pearlmutter, *Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code*, Tech. Rep. TR-ECE-08-01, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, 2008, Available at http://docs.lib.purdue.edu/ecetr/367.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] G.J. Sussman and G.L. Steele Jr., SCHEME*: an interpreter for extended lambda calculus*, A. I. Memo 349, MIT Artificial Intelligence Laboratory, 1975.

[41] G.J. Sussman, J. Wisdom, and M.E. Mayer, *Structure and interpretation of classical mechanics*, MIT Press, 2001.

[42] Y.M. Volin and G.M. Ostrovskii, *Automatic computation of derivatives with the use of the multilevel*

*differentiating technique — I: Algorithmic basis*, Computers and Mathematics with Applications 11 (1985), pp. 1099–1114.

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

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