Flattening Basic Blocks*

Jean Utke

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439–4844, USA; utke@mcs.anl.gov

1 The Problem

We consider a code that implements some numerical function

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$$

in the context of automatic differentiation (AD) via source-to-source transformation. Many AD-related publications [2, 3, 4, 5] assume the construction of the computational graph G representing either the entire $\boldsymbol{f}$, if possible, or at least an execution trace of $\boldsymbol{f}$ at a given argument. Following the familiar approach, $G = (V, E)$ is assumed to be a directed, acyclic graph (DAG). The set of vertices $V = X \cup Z \cup Y$ consists of vertices for the n independents X, vertices for the m dependents Y, and vertices for p intermediate values Z occurring in the computation of $\boldsymbol{f}$. The edges E represent the direct dependencies of the $w \in Z \cup Y$ computed with elemental functions $w = \phi(\ldots, v_i, \ldots)$ on the arguments $v_i \in X \cup Z$. The computations imply a dependency relation $v_i \prec w$ and its transitive closure $\prec^*$. The ϕ are the elemental functions (sin, cos, etc.) and operators (+,-,*, etc.) built into the given programming language. All edges $(v, w) \in E$ are labeled with the local partial derivatives $c_{wv} = \frac{\partial w}{\partial v}$. The graph G is the basis for numerous investigations into strategies for the efficient computation of derivative information, such as the Jacobian

$$\boldsymbol{J}(\boldsymbol{x}) = \left[\frac{\partial y_i}{\partial x_j}\right] \in \mathbb{R}^{m \times n}, i = 1, \ldots, m, j = 1, \ldots, n$$

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38, and by the National Science Foundation's Information Technology Research Program under Contract OCE-0205590, "Adjoint Compiler Technology & Standards" (ACTS).

and Jacobian-vector products $\boldsymbol{J}\dot{\boldsymbol{x}}, \boldsymbol{J}^T\bar{\boldsymbol{y}}$. As the exemplary application for this paper we take the computation of a Jacobian, namely the transformation of G into bipartite form [7]. Most practical applications do not require the complete $\boldsymbol{J}$ for $\boldsymbol{f}$. Instead, Jacobians $\boldsymbol{J}_k$ of subsections k of the code for $\boldsymbol{f}$ may be *preaccumulated*, for instance for use in a subsequent reverse sweep. Such $\boldsymbol{J}_k$ contain the partials of subsection outputs $\boldsymbol{y}_k$ with respect to their inputs $\boldsymbol{x}_k$. Therefore, depending on the application context, such a preaccumulation of sub-Jacobians can be beneficial if

- the preaccumulated Jacobian has fewer nonzero entries than intermediate values or partials that would need to be stored for a subsequent reverse sweep over all sections k, and
- computing $\boldsymbol{J}_k$ and (sparse) $\bar{\boldsymbol{x}}_k = \boldsymbol{J}_k^T\bar{\boldsymbol{y}}_k$ is cheaper than back propagation through the elemental ϕ.

One measure of the efficiency of the Jacobian computation is the number of operations incurred by the application of the chain rule seen as vertex [7] or edge elimination in G or as face elimination [12] in the corresponding directed line graph. The elimination order determines the operations count. Minimizing the operations count over all possible elimination orderings is conjectured to be an NP-hard problem. The search space size depends on the size of G. Most tools that implement AD algorithms use only the forward and reverse modes of AD. This strategy reduces the operations count minimization to a choice between the forward or the reverse elimination order. In particular, it eliminates the need to actually construct G. On the other hand, numerous examples show an advantage of the so-called cross-country elimination orderings over a strict forward or reverse mode. This is the main motivation to investigate the practical construction of G in an AD context. The computation of Jacobian vector products with a minimal Jacobian representation in the sense of scarcity preservation [8, 6] is another example of an application requiring G.

For the construction of G let us start with a simple example. When we look at a sequence of expressions of plain scalar variables, it appears intuitive how to concatenate expression graphs representing right-hand sides of assignments; we call this process *flattening* into a graph G.

In the example in Figure 1 we start by copying the right-hand-side expression graph of assignment a_1 into G and note the fact that `z` is the left-hand side by associating the maximal vertex $\ominus$ with `z`. We remember which vertex represents the argument `x`. These associations are shown as the thin dotted arrows. Even though `z` was also an argument, it was overwritten and therefore is associated only with the maximal vertex $\ominus$. Next we look at assignment a_2 and notice the use of `z` in the right-hand side. Clearly, this use has to be identified with the preceding left-hand side; that is, the `z` vertex in a_2 is identified with $\ominus$ already present in G. Furthermore we note `x` appearing as an argument again, and we identify it with the vertex in G that is already associated with `x`. Now that all arguments of a_2 have been identified in G, we

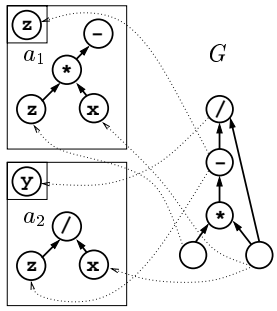

Fig. 1. Flattening of assignments a_1 and a_2 into G; a_1 : `z=-(z*x)`, a_2 : `y=z/x`

can simply copy the remaining vertex ⓪ (for the division intrinsic) and the attached two edges to G. The vertex ⓪ becomes the new maximal vertex and is associated with `y`, the left-hand side of a_2.

Obviously, practical codes are not restricted to scalars but use array dereferences, pointers, and so forth. For brevity we refer to such dereference expressions as "variables". The process described in the paragraph above relies on the identification of variables. In the case of plain scalar variables this amounts to a match of symbol and scope. In the general setting, however, we have to ensure that two variables are identified if they use the same address in memory. Assume all `z` in the example in Figure 1 are replaced by pointer dereferences `*z` and between a_1 and a_2 we find pointer arithmetic, for instance `z+=k` with some run-time parameter `k`. Without knowing `k` it is hard to tell what address `z` is pointing to. As shown in Figure 2, the right-hand-side to left-hand-side identification is no longer unique, indicated by the two dashed edges.

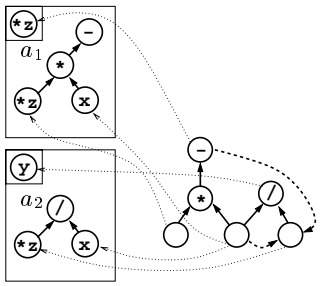

Fig. 2. For a_1 : `*z=-(*z*x)`, a_2 : `y=*z/x`, G becomes ambiguous.

In this paper we investigate the consequences of this ambiguity in the context of AD. Section 2 considers the choices for the scope of code subsections k for which we want to construct G. In Section 3 we formalize the flattening algorithm, and in Section 5 we show the solution to the ambiguity problem.

The construction and use of computational graphs for basic blocks also play a role in optimizing compiler technology. There are similarities in the approach [1] for constructing the graphs, but ambiguities may be treated differently. Their typical use for tasks such as register allocation and object code generation indicates a relatively advanced level in the compilation process that is far removed from the high-level programming language used for the source-to-source transformation in our context.[2] Consequently, we cannot rely on a compiler environment to provide the computational graphs but rather introduce an approach specifically suitable for the purposes of AD.

2 Control Flow Graph and the Scope of G

For programs implementing a general $\boldsymbol{f}$, the control flow may depend on the values of the arguments in X. For instance, argument-dependent loop bounds do not permit the construction of a single G representing the code for $\boldsymbol{f}$ for all possible input values. On the other hand, the construction of an argument-specific G, for example based on an execution trace, easily leads to huge computational graphs devoid of information indicating repeated structures. The actual goal of minimizing operations for the derivative computation sets practical limits for the size of G to which a minimization algorithm may be applied. Also, the minimization algorithms are generally too costly to be reapplied to each argument-specific G. Therefore we should require that G be structurally argument independent. Putting G in the context of the control flow graph (CFG) [1, 10] provides a simple criterion for structural independence. The vertices of a CFG are usually categorized into $\{Entry, Exit, Loop, EndLoop, Branch, EndBranch, BasicBlock\}$. For the following we assume a canonicalized representation of the program:

C1: All computations $w = \phi(\ldots, v_i, \ldots)$ are elements of an *assignment* statement a of the form $v := e$ with a single variable $v = lhs(a)$ on the left-hand-side and a right-hand-side expression $e = rhs(a)$ that is side-effect free.[3]

C2: All assignment statements are elements of an ordered list L of statements contained in a $BasicBlock$ vertex.

C3: A subroutine call is also a statement in L and may have side-effects; for instance, it may modify a global variable.

C4: All functions and operators that are not intrinsic or have side-effects are canonicalized into subroutine calls. Intrinsics have closed-form expressions for their partial derivatives, which is the case for all elementals ϕ.

C5: Intrinsics without arguments, for instance a function returning a constant value, are inlined.

[2] A low-level, compiler-integrated transformation is possible but, so far, has not been attempted in practice.

[3] Note, this excludes for instance C++ increment operations, as in `++i;` .

As indicated in Section 1, we consider computational subgraphs that either cover contiguous subsections of L of a single $BasicBlock$ or cover statements across $BasicBlock$ boundaries; see Section 7.1.

The parsing of the code implementing $\boldsymbol{f}$ by a compiler front-end yields the abstract syntax tree (AST), which contains the respective elements for assignments, their right-hand-side expression trees, subroutine calls, and so forth. The CFG is derived subsequently from the AST. It associates the sequence of assignments and subroutine calls in L with a $BasicBlock$.

The elemental operations $w = \phi(\ldots, v_i, \ldots)$ are the smallest computational subgraph. Because they already are bipartite, the first reasonable scope for subgraphs is the right-hand-side expressions. They are generally represented as *single-expression use* (seu) graphs, where every nonminimal vertex has at most one successor. In Section 3 we mention the intra right-hand-side variable identification that may lead to multiple successors for the minimal vertices. Except for parallel edges, the number of edges in a tree or seu graph representation is the same. Because of the special structure of seu graphs, the optimal elimination sequences can be constructed directly [11]. The so-called statement-level preaccumulation has been used in a number of AD tools. It yields noticeable gains over the plain forward mode in terms of computational cost.

However, we aim for an extension to the next level with G representing a sequence of right-hand-side expressions inside of a basic block. Going beyond a single right-hand-side expression implies loosing the seu property for G. Rather, G becomes a generic DAG. The motivation is that the extended scope yields a G that is larger but has more structural information than do the individual right-hand-side expressions. Minimizing over the larger G has the potential for a better solution than statement-level preaccumulation, which becomes just a special case. Staying in the scope of a basic block guarantees structural independence. Refer to Section 7.1 for the handling of subroutine calls.

3 Flattening Algorithm

We introduced the flattening in Section 1 for the example assignments shown in Figure 1. The formal algorithm follows the same principle. The association between vertices in G and vertices in the original assignments, in Figure 1 depicted by thin dotted lines, are represented as a list of vertex pairs. We iterate in an outer loop over all statements and for each statement we copy the respective right-hand-side expressions in a loop over all expression vertices and a subsequent loop over all expression edges. In order to find the proper endpoints for all edges in G we also need to maintain the association for all vertices representing intrinsics. This done with a second list of vertex pairs.

Algorithm 1 (Flattening Basic Blocks) *Consider a sequence $L' \subseteq L$ of assignments a in a basic block to be flattened into a directed acyclic graph $G =$*

(V, E). We maintain two tracking lists P_{var} (variables) and P_{intr} (intrinsics) of pairs (v_e, v_G) of vertices v_e from the expression $e = rhs(a)$ associated with vertices $v_G \in V$. The expression e itself is already given as a seu graph (V_e, E_e) by some compiler front-end. Perform the following steps:

$P_{var} := P_{intr} := \emptyset$
$V := E := \emptyset$
$\forall a \in L'$ *(in order)*
  $e := rhs(a)$
  $\forall v \in V_e$
    if $(\exists (v, .) \in P_{var})$ // v already represented in V
      do nothing
    elseif *((v is a variable)* $\vee$ $(|\{(w, v) : (w, v) \in E_e\}| > 0)$
      add new vertex v' *to* V*:*
        if (v is a variable) // not a constant or intrinsic
          $P_{var} := P_{var} + (v, v')$ // track v in the variable list
        elseif $(|\{(w, v) : (w, v) \in E_e\}| > 0)$ // must be an intrinsic
          $P_{intr} := P_{intr} + (v, v')$ // track v in the intrinsic list
        if $(\{(v, w) : (v, w) \in E_e\} = \emptyset)$
          $v'_{max} := v'$
  $\forall (v, w) \in E_e$
    add new edge (v', w') *to* G *where*
    $(v, v') \in P_{var} \cup P_{intr} \wedge (w, w') \in P_{intr}$
  if $(\exists (v, v') \in P_{var} : v = lhs(a))$ // if $lhs(a)$ is already tracked
    $P_{var} := P_{var} - (v, v')$ // remove it
  $P_{var} := P_{var} + (lhs(a), v'_{max})$ // track $lhs(a)$ as v'_{max}

Note that the algorithm has to keep G acyclic, a requirement that becomes an issue if a variable in any of the assignments in L' is used and then overwritten. Meeting this requirement is ensured by the last three lines of the algorithm where we always want to keep track of the "most recent" vertex in G that corresponds to a given variable. This tracking is sufficient because of canonicalizations C1 and C4. In Figure 1 we show the situation of `z` being overwritten. While processing a_1 we first add the pair `(z,`w`)` to P_{var}, where w is the left minimal vertex in G. After looping through all vertices and edges of $rhs(a_1)$ we find the left-hand-side `z` exists in P_{var}. We remove `(z,`w`)` and add `(z,`$\ominus$`)` instead.

The variable identification implied by the test $(\exists (v, .) \in P_{var})$ is described in Section 4. The algorithm is set up to identify input variables within and across right-hand-side expressions of assignments. Also note that it will only copy leaf nodes from the expression graphs that are variables. Intrinsics cannot be represented by minimal variables because of canonicalization C5 and constants always carry a zero edge label and therefore are ignored for our purposes. To limit the formalism, we exclude special cases such as purely constant assignments.

4 Variable Identification via Alias Analysis

We already mentioned the need for variable identification by address for pointer dereferences, array subscripting, and so forth. In a compiler context, equivalence information over (virtual) addresses is provided by *alias analysis* [10]. The example in Figure 2 shows that syntactic equivalence is not sufficient for identification; instead, one must ensure that `z` points to the same address in both cases. The same applies to array subscripting, which is ubiquitous in numerical codes. Therefore we have to identify variables by means other than syntactic equivalence, for instance by (virtual) address equivalence.

We distinguish *flow-sensitive* and *flow-insensitive* as well as *must* and *may* alias analysis; their respective results are given in a variety of formats. For the purpose of this paper we introduce a vector of virtual address sets A. The use u_v of a variable v at a specific point in the code refers to a particular set A_{u_v}, through an index, which allows one to represent flow-sensitive analysis results. Flow-insensitive alias analysis simply refers to the same address set in the alias vector for every use of v in the entire program. If A_{u_v} contains exactly one address, it expresses must-alias information; that is, v must use that one given address. If A_{u_v} contains more than one address, it represents may-alias information; that is, v may use one of the given addresses.[4] For instance, whenever there is a dependence of address calculations on run-time parameters, the alias analysis will be unable to narrow A_{u_v} down to a single address.

Figure 3 shows an example with indices into the alias vector depicted in the grey ovals. The conservatively assumed default lets all variables be aliased to everything, for example by referring to a special entry `<all>` in the alias vector representing the entire address space.

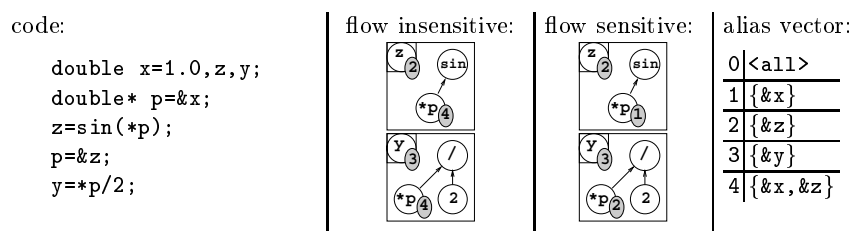

Fig. 3. Variable uses and address sets for a simple C code

In order to positively identify vertex w with v, the test $(\exists(v,.) \in P_{var})$ has to be rephrased in terms of address sets:

1. $|A_{u_w}| = 1$; that is, u_w occupies a single, unambiguous address.
2. $\exists(u_v,.) \in P_{var}$ with $A_{u_v} = A_{u_w}$; that is, u_v and u_w must share the same address.

[4] but it has to use one of them

For the opposite test that w cannot be positively identified with any $(v, .) \in P_{var}$, it suffices to check $\forall (v, .) \in P_{var} : A_{u_w} \cap A_{u_v} = \emptyset$.

5 Removing Ambiguity by Splitting

If in Algorithm 1 we cannot identify a given $v \in V_e$ with any $w \in V$, then the algorithm creates a new vertex v' in G. This already applies to variables within a right-hand-side expression; see Figure 4 (a,b) for an example expression `*q+*r+*s`. The dashed edges indicate may-aliasing, which in the context of

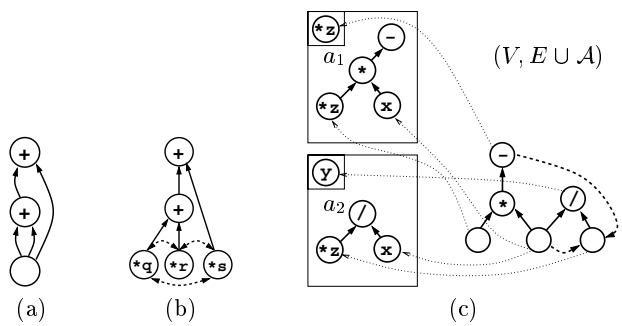

Fig. 4. Variable identification: intra right-hand side with (a): $|A_{u_{*\mathtt{q}}}| = 1$, $A_{u_{*\mathtt{q}}} = A_{u_{*\mathtt{r}}} = A_{u_{*\mathtt{s}}}$, (b): $|A_{u_{*[\mathtt{q},\mathtt{r},\mathtt{s}]}}| > 1, A_{u_t} \cap A_{u_{t'}} \neq \emptyset, t, t' \in \{*\mathtt{q}, *\mathtt{r}, *\mathtt{s}\}$, and (c): ambiguous G as in Figure 2 with $|A_{u_{*\mathtt{z}}}| > 1, A_{u_\mathtt{x}} \subset A_{u_{*\mathtt{z}}}$

G can be expressed with edges $\mathcal{A} = \{(v, w) \in G : A_v \cap A_w \neq \emptyset\}$. The order of the sequence of assignments $L' = (a_1, \ldots, a_p)$ that is the input to Algorithm 1 is essential for the semantic of the basic block. Algorithm 1 replaces this order by the evaluation order imposed by the directed edges in G. If $\mathcal{A} = \emptyset$, then G has minimal vertex count; it represents the dependency information exactly and thereby preserves semantics. $\mathcal{A} \neq \emptyset$ in Figure 4 (b) is benign; there are merely more vertices in the graph than in (a). $\mathcal{A} \neq \emptyset$ in Figure 4 (c) has ambiguous dependency information. To preserve semantics through correct dependency information, we therefore need to ensure that the subset $\hat{\mathcal{A}} = \{(v, w) \in \mathcal{A} : w \in rhs(a_i) \wedge v = lhs(a_k), k < i\} = \emptyset$. $(V, E \cup \hat{\mathcal{A}})$ is a set of possible graphs G. We can, however, not decide which G is the semantically correct one.

5.1 Splitting into Edge Subgraphs

We define an *edge subgraph* $G_s = (V_s, E_s)$ of a graph $G = (V, E)$ with $V_s \subseteq V$ and $E_s \subseteq E$ such that if $(v, w) \in E_s$, then $v, w \in V_s$, and if $(t, u), (v, w) \in E_s \wedge (u, v) \in E$, then $(u, v) \in E_s$. A *split* of G into edge subgraphs $G_i = (V_i, E_i)$

is defined such that $(E = \bigcup E_i) \wedge (E_i \cap E_j = \emptyset)$. A split duplicates vertices v for all pairs $(u,v),(v,w) \in E \wedge (u,v) \in E_i \wedge (v,w) \in E_j$ such that they occur in both graphs G_i and G_j. The identity between vertices $v = v_i$ in G_i and $v = v_j$ in G_j can be expressed with the set of (virtual) identity edges $\mathcal{I} = \{(v_i, v_j)\}$. In the graph $(\bigcup V_i, \bigcup E_i \cup \mathcal{I})$ we find all G_i and the pairs of duplicated vertices connected by the edges in $\mathcal{I}$. With these identities one can view the splitting into edge subgraphs as the inverse of Algorithm 1.

Consider all edges $(v,w) \in \hat{\mathcal{A}}$ as possible identities like those in $\mathcal{I}$. In $(V, E \cup \hat{\mathcal{A}})$ for each such w only one (or no) element of $\hat{\mathcal{A}}$ is the actual identity; that is, $(v,w) \in \mathcal{I}$. Like $(\bigcup V_i, \bigcup E_i \cup \mathcal{I})$ we view $(V, E \cup \hat{\mathcal{A}})$ as consisting of edge subgraphs G_i that

1. have $\hat{\mathcal{A}}_i = \emptyset$, that is, locally unambiguous dependency information, and
2. can be (partially) ordered with "$\prec$" such that $\forall (v,w) \in \hat{\mathcal{A}} : v \in G_j$ then $w \in G_k, G_j \prec G_k$, and, vice versa, $\forall (t,u) \in \hat{\mathcal{A}} : u \in G_j$ then $t \in G_i, G_i \prec G_j$.

In other words all virtual in-edges are out-edges of preceding graphs and all virtual out-edges are in-edges of succeeding graphs. Figure 5 shows $(V, E \cup \hat{\mathcal{A}})$ corresponding to Figure 2 with two edge subsets as shaded areas on the left. The split results in the two shaded, unambiguous subgraphs on the right; the connecting virtual edge imposes the execution order. The ordering between

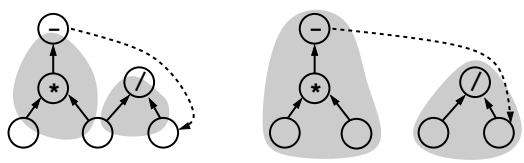

Fig. 5. $(V, E \cup \hat{\mathcal{A}})$ for the example in Figure 2 before (left) and after (right) split

the graphs preserves the semantics. The splitting criterion implies a minimal number of subgraphs, but it does not determine the subgraphs G_i uniquely. For example, consider $\hat{\mathcal{A}} = \{(v,w),(v',w)\}$ resulting in G_1 and G_2. Any edge (t,u) for which there are no paths $P_{u,v}, P_{u,v'}, P_{w,t}$ can be made part of either G_1 or G_2. More generally we can define the set of edges that are *movable* between edge subgraphs G_i and G_j as $\{(t,u) \in E : \forall (v,w) \in \hat{\mathcal{A}} : v \in V_i \wedge w \in V_j : \nexists\, P_{u,v}, P_{w,t}\}$. Consequently, one can formulate criteria to determine an optimal split that we will briefly investigate in Section 7.2.

5.2 Determining Jacobian Entries

We now have a split into l unambiguous subgraphs $G_1, \ldots, G_l$. Before performing any elimination in G_i we have to determine which vertices in G_i are independent and which are dependent in order to obey the restrictions on

vertex and edge eliminations or build the proper directed line graph for face elimination, respectively. Obviously, the set of independents is exactly the set of n_i minimal vertices $\{v : \nexists\,(u,v) \in E_i\}$. However, the usual assumption that all maximal vertices constitute the dependent set is not necessarily true. One might simply have a variable v assigned that is then referred to in a right-hand-side expression flattened into G_i as well as one flattened into a successor G_j. If we knew $\mathcal{I} \subseteq \hat{\mathcal{A}}$ exactly, we could determine the m_i dependents as $\{v : (v,w) \in \mathcal{I}, v \in V_i, w \in V_j, j > i\} \cup \{v : \nexists\,(v,u) \in E_i\}$. Then the elimination in G_i yields $\boldsymbol{J}_{G_i} \in \mathbb{R}^{m_i \times n_i}$. If L' in Algorithm 1 contains the entire code subsection k of interest (see Section 1), then we can write

$$\boldsymbol{J}_k = \prod_{i=1}^{l} \left(\boldsymbol{P}_i^{(r)} \begin{bmatrix} \boldsymbol{J}_{G_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_i \end{bmatrix} \boldsymbol{P}_i^{(c)} \right)$$

with the identity $\boldsymbol{I}_i \in \mathbf{R}^{s_i \times s_i}, s_i = |\{(v,w) \in \mathcal{I}, v \in V_j, w \in V_{j'}\ j < i < j'\}|$. $\boldsymbol{P}_i^{(r)}, \boldsymbol{P}_i^{(c)}$ are permutation matrices that line up the rows and columns correctly. $\mathcal{I}$ is not known, a conservatively correct approach is to consider all all maximal vertices and all vertices with virtual out-edges in $\hat{\mathcal{A}}$ dependent. This is obviously suboptimal, and a better and practical solution is described in the following section. While we can write the above formula for $\boldsymbol{J}_k$, the $\boldsymbol{P}_i^{(r)}, \boldsymbol{P}_i^{(c)}$, and $\boldsymbol{I}_i$ are fully determined only at runtime. A practical approach would generate code containing saxpy operations over the elements of the $\boldsymbol{J}_{G_i}$, implementing a sequence of sparse Jacobian vector products.

6 Practical Solution

Disregarding the option for optimizing the split into subgraphs, we can pick a convenient splitting point such as splitting exactly along the assignments a in L'. This entails a number of simplifications, in particular the ability to use the alias information in a suitably enhanced format known as *du/ud-chains* [10]. In short, a ud-chain (read use-define-chain) D_{u_v} contains for a particular use of a variable v the locations of possible definitions, that is, assignments to v. Similarly, a du-chain (read define-use-chain) U_{u_v} contains for a particular definition of v the locations of all possible uses. Because of canonicalizations C1, C2, and C4 we can simply equate these locations with the statements in the *BasicBlock* vertices. Traditionally, ud-chains are introduced for the use of a variable following alternative definitions in separate branches in the control flow. If a ud-chain for v refers to exactly one statement, then this is the most recent assignment to v with respect to the control flow and $|A_{u_v}| = 1$. If $|A_{u_v}| > 1$, then the chain may refer to more than one statement even in the same *BasicBlock*. In principle there is no limit to the locations du/ud-chains may refer to. Dereferencing a global pointer variable can entail chains referring to locations all over the code. Limiting the scope of the flattening algorithm,

we can reduce the needed information from ud/du-chains to statements within the scope and a placeholder "0" for defining locations outside the scope.

6.1 Graph-Splitting Algorithm

The following algorithm is a somewhat simplified version of the practical implementation mentioned in Section 7.

Algorithm 2 (Semantic-Preserving Flattening) *Consider a sequence of assignments $L = (a_1, \ldots, a_l)$ to be flattened into an ordered sequence of directed acyclic graphs $G_i = (V_i, E_i)$. We maintain two tracking lists P_{var} (variables) and P_{intr} (intrinsics) of pairs (v_e, v_{G_i}) of vertices v_e from the expression $e = rhs(a)$ associated with vertices $v_{G_i} \in V_i$. Perform the following steps.*

init: $i := 0$
$k := k' := 1$
split: $i := i + 1; \quad k' := k$
$G_i := (V_i := \emptyset, E_i := \emptyset)$
$P_{var} := P_{intr} := S_{lhs} := \emptyset$
loop: $e := rhs(a_k)$
$\forall v \in V_e$
  if (v is a variable)
    if $[(D_{u_v} = (a_j) \wedge j < k') \bigwedge$ //defined outside of G_i and
      $(\nexists\, (w, .) \in P_{var} : D_{u_v} = D_{u_w})]$ //not already there
      $\bigvee\, (D_{u_v} = (0)))$ //or defined outside of the scope
      add new vertex v' *to* V_i; $P_{var} := P_{var} + (v, v')$
    if $(|D_{u_v}| > 1 \wedge \exists a_j \in D_{u_v} : j \geq k')$ //ambiguous
      remove all additions to G_i *done for* a_k
      goto split:
  elseif $(|\{(w, v) : (w, v) \in E_e\}| > 0)$ //must be an intrinsic
      add new vertex v' *to* V_i; $P_{intr} := P_{intr} + (v, v')$
$\forall (v, w) \in E_e$
  add new edge (v', w') *to* E_i *where*
  $((v, v') \in P_{var} \cup P_{intr}) \vee ((t, v') \in P_{var} \wedge D_{u_v} = D_{u_t}) \bigwedge (w, w') \in P_{intr}$
if $(\exists (w, w') \in P_{var} : A_{u_w} \cap A_{u_v} \neq \emptyset$ *with* $v = lhs(a_k))$
  $P_{var} := P_{var} - (w, w')$
$P_{var} := P_{var} + (lhs(a_k), v'_{max})$
$S_{lhs} := S_{lhs} \cup lhs(a_k)$
if $(k < |L|)$
  $k := k + 1$
  goto loop:
else
  done

In the algorithm the first statement in each G_i has index k'. This allows one to distinguish definitions of variables inside or outside of the currently considered G_i. Reducing the ud-chain information to the given subsection scope has

the drawback that variables with the same outside-of-scope definition cannot be identified and, while preserving semantical correctness, we do not achieve the minimal vertex count. Like Algorithm 1 it ignores constants. If we take the example in Figure 2 and assume for a_1 that $D_{u_{*z}} = D_{u_x} = (0)$ and for a_2 that $D_{u_{*z}} = (a_1, 0), D_{u_x} = (0)$, then the algorithm will return G_1 and G_2 equivalent to the right-hand-sides of a_1 and a_2, respectively. If, however, in a_2 we have $D_{u_{*z}} = (a_1)$ then the algorithm returns the equivalent of G as shown in Figure 1.

PROPOSITION: Algorithm 2 attains the minimal number of subgraphs satisfying the criterion in Section 5.1. *Proof:* A split into two subgraphs $G_1 = (v_1, E_1), G_2 = (v_2, E_2)$ with $v \in V_1, w \in V_2$ can cover all virtual edges (v', w') in a set S with $S = \{(v, w).(v', w') \in \hat{\mathcal{A}} : \nexists\, P_{v,w'}, P_{w,v'} \in (V, E \cup \hat{\mathcal{A}})\}$. If $\exists P_{w,v'}$, then v and w' and by edge subgraph definition all vertices and edges on the path would have to be in one subgraph. Each virtual edge belongs to such an S and $(V, E \cup \hat{\mathcal{A}})$ defines s sets. Then the minimal number of subgraphs is $s + 1$. The algorithm keeps flattening into the same subgraph as long as all encountered virtual edges (v, w) originate outside of G_i in another G_j (this does not actually consider outside-of-scope references as edges). All targets w lie inside of G_i. Algorithm 2 creates a new subgraph whenever it encounters the target vertex of an edge $(v', w') \in \hat{\mathcal{A}}$ with a source with G_i. That means a $P_{w,v'}$ exists and a (v', w') belongs to a new equivalence set.□

6.2 Determining the Dependents

The collected S_{lhs} for each G_i can be used in conjunction with du-chains to narrow down the proper set of dependent variables. Prior to the first statement in *split* the set of dependent variables for G_i is defined as

$$\{v \in S_{lhs} : \exists a_j \in U_{u_v} : j > k \bigvee 0 \in U_{u_v}\} \quad .$$

Because of ambiguity this is in general a superset of the exact set of dependent variables but is in any case a subset of S_{lhs}. Similarly, all dependent variables $\boldsymbol{y}_k$ of code subsection k having the same scope as the du-chain information are determined by all variables v with $0 \in U_{u_v}$.

We mentioned in Section 5.1 that the dependents may not be the maximal vertices. In order to perform a complete vertex elimination yielding a bipartite graph, the graph with nonmaximal dependent vertex v needs to be augmented with vertex v' and an edge (v, v') that has an edge label $c_{v'v} = 1$. Then v' takes the place of v in the dependent set, and a complete vertex elimination is possible. In the case of edge eliminations we have to make sure that there are no edge-front eliminations performed on in-edges of v. On the other hand, there are no consequences for face elimination as the construction of the directed line graph [12] properly represents v; see Figure 6 (a) and (b).

As the flattening algorithm identifies the left-hand-side variables with the respective maximal right-hand-side expression vertices, an assignment $y = x$

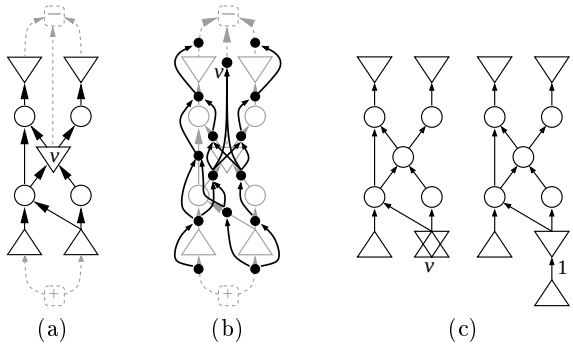

Fig. 6. Nonmaximal dependents v: (a) extended G, (b) overlaid directed line graph, (c) collapsed (left) and separated (right) independent/dependent variable

can collapse an independent and a dependent variable into the same vertex. In G this may lead to a single unconnected vertex or a situation depicted in the left graph in Figure 6 (c). In both cases a split of the collapsed vertex into two vertices with a connecting edge solves the issue. The unit edge label is the corresponding Jacobian entry.

7 Outlook and Conclusions

An extended version of Algorithm 2 is implemented in the OpenAD[5] framework of the Adjoint Compiler Technology & Standards (ACTS) project. For the ACTS project the aforementioned alias analysis, the generation of du/ud-chain information, and other analyses are being implemented in OpenAnalysis.[6] The implementation cannot rely on all canonicalizations required in Section 2; it also has to incorporate the handling of the special cases mentioned in Section 6.2. While this complicates the case distinction in the algorithm, it does not change its basic functionality. Also not covered in this paper is a major addition to the implemented algorithm that integrates an intrinsic specific activity analysis with the flattening. While the algorithms described here only ignore constant vertices, there are some intrinsics such as `floor` that are constant in open subdomains. Compiler style activity analysis implemented as a dataflow analysis is typically not concerned with this level of detail. OpenAD considers such intrinsics and can recognize ensuing constant subgraphs across assignment boundaries. Since constant subgraphs do not need to be flattened into G, we do not need unique variable identification for the constant propagation. Instead, we can establish a variable v is constant if we found all assignments occurring in D_{u_v} to be constant as well. The aspects mentioned in the following sections represent possible extensions.

[5] `www-unix.mcs.anl.gov/~ utke/OpenAD`

[6] `www.hipersoft.rice.edu/openanalysis.`

7.1 Extending the Scope

While we concentrated on flattening of assignments in basic blocks, there are some obvious ways of considering a computational graph in scopes across the boundaries of a single basic block, which are

1. exploiting interface contraction
2. inlining subroutine calls, and sequentializing branches.

The first application is of clear practical value, the method and the benefits are described in [9].

Building a computational graph through the body of a subroutine that is being called from the code subsection in question amounts to inlining. In a source transformation context inlining should be done explicitly by a compiler front-end prior to any AD code transformation. On the other hand, we can flatten "black box" subroutine calls if we can obtain all partial derivative values directly, for instance, if they are returned as a specific set of parameters. The flattening algorithm can treat such subroutine calls just like intrinsics. Note that in a ud-chain, simply referring to the subroutine call as the definition location for a variable is no longer sufficient because the subroutine may define multiple variables and we then need additional information to find out which one.

The last extension, the sequentializing of branches, has limited practical value if the computational cost of executing all branches rather than one becomes too high. The source transformation algorithm will have to make the branches mutually independent, execute them sequentially, and select the proper result. With alias information and du/ud-chains we already have the prerequisites for such a transformation. It is, however, beyond the scope of this paper.

7.2 Optimizing the Split

In theory, because of the edges that are movable between subgraphs G_i (see Section 5.1), we can have a variation of the number of independent and dependent vertices in the G_i, a variation in the sparsity of the J_{G_i}, and, as the G_i change, a variation of the minimal cost for preaccumulating the corresponding J_{G_i}. The first two obviously affect the storage requirements for a subsequent reverse sweep and the cost of the implied Jacobian vector products. So far we have considered only minimizing the preaccumulation cost, which now would become the inner problem of a nested optimization. At this point we are not able to tackle such a nested optimization in practice and therefore stay with the simple split choice made for Algorithm 2.

Moreover, one can construct examples showing that introducing splits additional to the minimally necessary ones can undercut the above minimization criteria. A simple example is a sequence of $\boldsymbol{x} = \boldsymbol{a}\boldsymbol{b}^T\boldsymbol{x}; \boldsymbol{x} = \boldsymbol{a}\boldsymbol{b}^T\boldsymbol{x}; \ldots$ implemented by $z = \boldsymbol{b}^T\boldsymbol{x}; \boldsymbol{x} = \boldsymbol{a}z$ and assuming splits necessitated by an inability

to identify $\boldsymbol{x}$ between the left- and right-hand sides. If we also split through z, then instead of storing n^2 Jacobian entries from $\boldsymbol{a}\boldsymbol{b}^T$, we can store $\boldsymbol{a}$ and $\boldsymbol{b}$ with $2n$ entries. Rather than introducing additional splits, this issue should be seen in the context of scarcity-preserving eliminations [8, 6].

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
2. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.
3. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
4. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.
5. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.
6. A. Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003.
7. A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, 1991.
8. A. Griewank and O. Vogel. Analysis and exploitation of Jacobian scarcity. In *Proceedings of HPSC Hanoi*. Springer, 2003. To appear.
9. P. Hovland, C. Bischof, D. Spiegelman, and M. Casella. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. *SIAM Journal of Scientific Computing*, 18,4:1056–1066, 1997.
10. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.
11. U. Naumann. Optimal pivoting in tangent-linear and adjoint systems of nonlinear equations. Preprint ANL-MCS/P944-0402, Argonne National Laboratory, 2002.
12. U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 3(99):399–421, 2004.