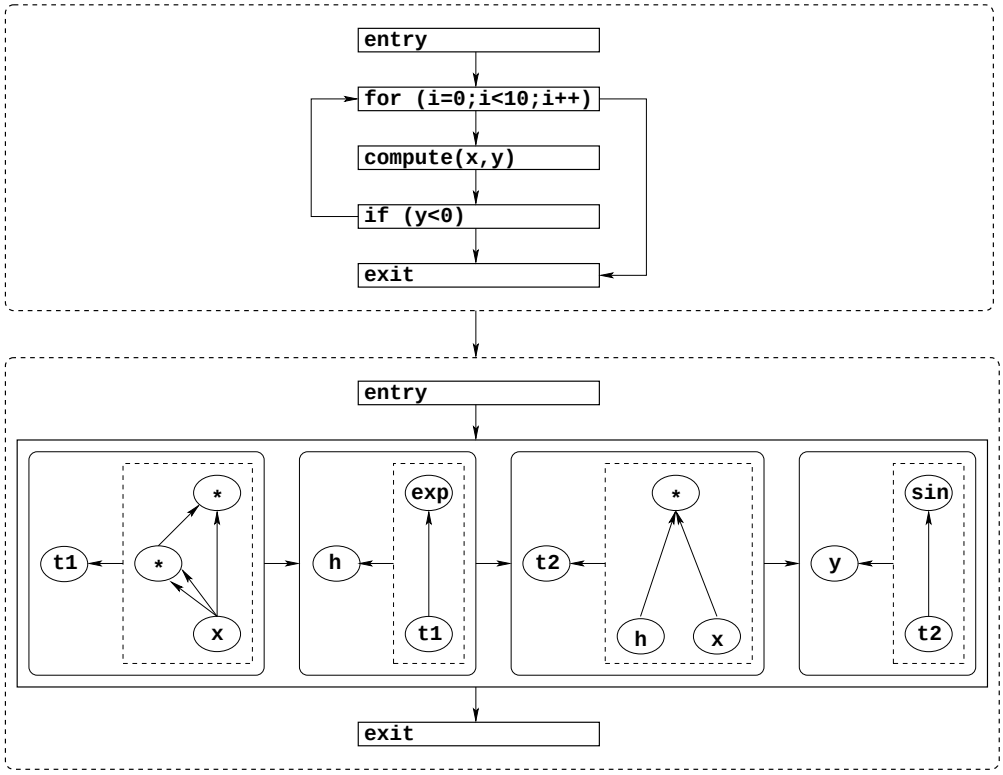

Figure 1: XAIF Graph Hierarchy

1 Example

In this section we show how the XAIF is used in the automatic differentiation of the following C code. The complete example, including the XAIF for the original and transformed source code, is available at the XAIF web page.

```
void head(double x, double y) {
  int i;
  for (i=1;i<10;i++) {
    compute(x,y);
    if (y<0) exit;
  }
}

void compute(double x, double y) {
  double h;
  h=exp(x*x*x);
  y=sin(h*x);
}
```

1.1 XAIF of Original Program

In XAIF, a program is represented as a hierarchy of directed graphs, as shown in figure 1. The call graph consists of two vertices representing the two subroutines `head` and `compute`. The call of `compute` inside `head` is represented by the edge connecting these vertices.

```
<?xml version="1.0" encoding="UTF-8"?>
<xaif:CallGraph ... >
  <xaif:CallGraphProperties>
    ...
  </xaif:CallGraphProperties>
  <xaif:SymbolTable>
    ...
  </xaif:SymbolTable>

  <!-- head(double x, double y) -->
  <xaif:CallGraphVertex id="0" symbolId="head">
    ...
  </xaif:CallGraphVertex>

  <!-- void compute(double x, double y) -->
  <xaif:CallGraphVertex id="1" symbolId="compute">
    ...
  </xaif:CallGraphVertex>

  <xaif:CallGraphEdge id="0" source="0" target="1"/>

</xaif:CallGraph>
```

The control flow graph of `head` contains three vertices in addition to the standard `entry` and `exit` vertices. If the `for`-loop condition is satisfied, the loop body gets executed. Otherwise, the program is continued with the first statement following the loop. In this example no statement follows the loop, which results in an edge leading into the `exit` vertex.

The first statement inside the loop body is the call of `compute` followed by an `if`-statement. Depending on the value of the test, the loop is exited or the next statement in the loop body is executed. As the `if`-statement happens to be the last statement of the loop body, this is equivalent to jumping back to the head of the loop.

```
<xaif:ControlFlowGraph>
  <xaif:ControlFlowGraphProperties>
    ...
  </xaif:ControlFlowGraphProperties>

  <xaif:SymbolTable>
    ...
  </xaif:SymbolTable>

  <xaif:ControlFlowVertex id="0" name="Entry"/>
  <xaif:ControlFlowVertex id="1" name="ForLoop">...</xaif:ControlFlowVertex>
  <xaif:ControlFlowVertex id="2" name="BasicBlock">...</xaif:ControlFlowVertex>
  <xaif:ControlFlowVertex id="3" name="If">...</xaif:ControlFlowVertex>
  <xaif:ControlFlowVertex id="4" name="Exit"/>

  <xaif:ControlFlowEdge id="0" source="0" target="1"/>
  <xaif:ControlFlowEdge id="1" source="1" target="4"/>
  <xaif:ControlFlowEdge id="2" source="1" target="2"/>
  <xaif:ControlFlowEdge id="3" source="2" target="3"/>
  <xaif:ControlFlowEdge id="4" source="3" target="1"/>
  <xaif:ControlFlowEdge id="5" source="3" target="4"/>
</xaif:ControlFlowGraph>
```

The control flow inside `compute` is straightforward. It consists of a single basic block in addition to `entry` and `exit`. After canonicalization (performed by the front-end), the basic block contains four assignment statements, which are represented by the four vertices of the `BasicBlockGraph` element shown below.

```
...
<xaif:BasicBlockGraph>
  <xaif:BasicBlockGraphProperties>
    <xaif:Property id="0" name="inloop" value="no"/>
  </xaif:BasicBlockGraphProperties>

  <!-- t1 = x*x*x; -->
  <xaif:BasicBlockVertex id="0" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- h = exp(t1); -->
  <xaif:BasicBlockVertex id="1" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- t2 = h*x; -->
  <xaif:BasicBlockVertex id="2" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- y = sin(t2); -->
  <xaif:BasicBlockVertex id="3" name="AssignmentStatementGraph">
    ...
  </xaif:BasicBlockVertex>

  <!-- data flow -->
  <xaif:BasicBlockEdge id="0" source="0" target="1"/>
  <xaif:BasicBlockEdge id="1" source="1" target="2"/>
  <xaif:BasicBlockEdge id="2" source="2" target="3"/>
</xaif:BasicBlockGraph>
...
```

Each `AssignmentStatementGraph` consists of a variable reference representing the left-hand side and some expression DAG representing the right-hand side, as illustrated next.

```
...
<!-- t2 = h*x; -->
<xaif:BasicBlockVertex id="2" name="AssignmentStatementGraph">
  <xaif:AssignmentStatementGraph>
    <xaif:VariableReferenceVertex id="0" symbolId="1_4"/>
    <xaif:AssignmentRHSVertex id="1">
      <xaif:ExpressionGraph>
        <xaif:VariableReferenceVertex id="0" symbolId="1_3"/>
        <xaif:VariableReferenceVertex id="1" symbolId="1_1"/>
        <xaif:BinaryExpressionVertex id="2" name="Multiply"/>
        <xaif:ExpressionEdge id="0" source="0" target="2"/>
        <xaif:ExpressionEdge id="1" source="1" target="2"/>
      </xaif:ExpressionGraph>
    </xaif:AssignmentRHSVertex>
    <xaif:AssignmentStatementEdge id="0" source="1" target="0"/>
  </xaif:AssignmentStatementGraph>
</xaif:BasicBlockVertex>

<!-- y = sin(t2); -->
<xaif:BasicBlockVertex id="3" name="AssignmentStatementGraph">
  <xaif:AssignmentStatementGraph>
    <xaif:StatementProperties>
    </xaif:StatementProperties>
    <xaif:VariableReferenceVertex id="0" symbolId="1_5"/>
    <xaif:AssignmentRHSVertex id="1">
      <xaif:ExpressionGraph>
        <xaif:SubroutineCallExpressionVertex id="0" symbolId="exp">
          <xaif:SubroutineArgument>
            <xaif:VariableReference symbolId="1_4"/>
          </xaif:SubroutineArgument>
        </xaif:SubroutineCallExpressionVertex>
      </xaif:ExpressionGraph>
    </xaif:AssignmentRHSVertex>
    <xaif:AssignmentStatementEdge id="0" source="1" target="0"/>
  </xaif:AssignmentStatementGraph>
</xaif:BasicBlockVertex>
...
```

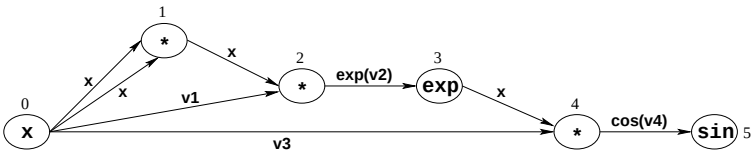

Figure 2: Linearized Computational Graph

The minimal vertices in the expression DAGs represent variable references. All other vertices are arithmetic operations or function calls. The value associated with the maximal vertex is assigned to the variable referenced on the left-hand side. Intermediate results are labeled $v1, \ldots, v5$.

1.2 XAIF of Transformed Program

In this section we describe the result of transforming the original program semantically according to a two-step AD algorithm:

1. Derivative code for computing the local Jacobian of the basic block in `compute` is generated. Since in this simple example there are just one independent `x` and one dependent `y` variable, the local Jacobian contains only one element $\partial \mathtt{y}/\partial \mathtt{x}$, which is the derivative of `y` with respect to `x`.

2. Derivative code generated according to the rules of forward-mode AD computes directional derivatives of the dependent with respect to the independent variables, that is, Jacobian matrix times vector products. In the example, this simplifies to a weighted derivative, namely, the product of $\partial \mathtt{y}/\partial \mathtt{x}$ and some scalar weight `ad_x`. The semantics of the program is changed in order to compute this value.

Referencing relevant literature, we briefly discuss a method for generating optimal derivative code for the local Jacobian of `compute`. The main elements of the corresponding XAIF representation are presented in the context of the XAIF of the forward-mode AD transformed code.

The computational graph of the basic block is shown in Figure 2. Expressions for the local partial derivatives are attached to the edges. Given a value for `x`, one can evaluate these expressions during a single evaluation of the basic block in parallel with the actual function value `y` itself. This results in a linearized version of the computational graph. As shown in [5], the value of derivative $\partial \mathtt{y}/\partial \mathtt{x}$ at the current argument can be accumulated by eliminating the p intermediate vertices in the linearized computational graph (in our example, $p = 4$). The order in which this is done determines the number of scalar floating-point operations required for this process. Minimizing this value over the $p!$ different elimination orderings is conjectured to be an NP-complete [4] combinatorial optimization problem [2, 5].

A deterministic algorithm for gradients with single-read intermediate variables (such as the graph in our example) is discussed in [6]. It leads to the following derivative code for `compute`:

```
void compute_ad(double x, double ad_x, double y, double ad_y) {

  double h;
  double t1, t2;
  double _dy_dx_t1,_dy_dx_t2,_dy_dx_t3,_dy_dx_t4;
  double _dy_dx;

  t1=x*x*x;
  h=exp(t1);
  t2=h*x;
  y=sin(t2);

  _dy_dx_t1 = (x+x)*x+x*x
  _dy_dx_t2 = exp(t1)
  _dy_dx_t3 = _dy_dx_t1*_dy_dx_t2*x+h
  _dy_dx_t4 = cos(t2)
  _dy_dx = _dy_dx_t3*_dy_dx_t4

  ad_y = _dy_dx * ad_x
}
```

The computation of `_dy_dx_t1` corresponds to the elimination of vertex `1` in the linearized computational graph. Vertices `2` and `3` are eliminated by computing `_dy_dx_t3`. Finally, the elimination of vertex `4` leads to `_dy_dx`, which represents the pre-accumulated value of $\partial \mathtt{y} / \partial \mathtt{x}$. The subroutine itself is transformed into a semantically different version `compute_ad` with inputs `x` and `ad_x` and outputs `y` and `ad_y`. It is straightforward to verify that for a given argument `x` and a derivative weight `ad_x`, `compute_ad` computes both the function value `y` and `ad_y`—the directional derivative of `y` with respect to `x` in direction `ad_x`. This is what we expect from a forward-mode AD-transformed derivative code. The XAIF of `compute_ad` is analogous to the one for the original routine, with a basic block containing additional assignments and several new entries in the symbol table and argument list.

The forward-mode AD version of the top-level routine `head` is as follows.

```
void head_ad(double x, double ad_x, double y double ad_y) {
  int i;
  for (i=1;i<10;i++) {
    compute_ad(x,ad_x,y,ad_y);
    if (y<0) exit;
  }
}
```

The control flow remains unchanged: `head_ad` calls `compute_ad` to compute both `y` and `ad_y` for given `x` and `ad_x`. Again, the XAIF is analogous to the one for the `head` subroutine with the necessary changes or additions made to the symbol table, argument list, and call to `compute`. The XAIF of the entire derivative code can be found at `www.mcs.anl.gov/xaif`.

References

[1] BERZ, M., BISCHOF, C., CORLISS, G., AND GRIEWANK, A., Eds. *Computational Differentiation: Techniques, Applications, and Tools* (1996), Proceedings Series, SIAM.

[2] BISCHOF, C., AND HAGHIGHAT, M. Hierarchical approaches to Automatic Differentiation. In *[1]* (1996), SIAM, pp. 82–94.

[3] CORLISS, G., AND GRIEWANK, A., Eds. *Automatic Differentiation: Theory, Implementation, and Application* (1991), Proceedings Series, SIAM.

[4] GAREY, M., AND JOHNSON, D. *Computers and Intractability - A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, San Francisco, 1979.

[5] GRIEWANK, A., AND REESE, S. On the calculation of Jacobian matrices by the Markovitz rule. In *[3]* (1991), SIAM, pp. 126–135.

[6] NAUMANN, U. On optimal Jacobian accumulation for computations with single use of intermediate variables. Tech. Rep. Preprint ANL/MCS-P944-0402, 2002.