Generating adjoint expressions for Matlab

Johannes Willkomm

Institute of Scientific Computing
RWTH Aachen University

Tenth European Workshop on Automatic Differentiation
Outline

1. Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2. Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3. Solution
   - Recursive construction
   - XSLT implementation

4. Results and Conclusion
   - An example
   - Conclusion
Outline

1 Motivation
   - Generating adjoint code for Matlab
     - Scalar adjoint rules are not enough
2 Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization
3 Solution
   - Recursive construction
   - XSLT implementation
4 Results and Conclusion
   - An example
   - Conclusion
ADiMat implements AD *source transformation* of Matlab code. Consider a function with signature `function z = f(a)`

- **Forward mode:** `adimat f.m` produces
  ```matlab
  function [g_z, z] = g_f(g_a, a)
  ```
- **Reverse mode:** `admproc f.m` produces
  ```matlab
  function [a_a, z] = a_f(a, a_z)
  via XSL transformations
  ```

In both cases the derivative variables can be either native doubles, which (in general) results in scalar mode or one can use one of several derivative classes, which allows vector mode.
Outline

1 Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2 Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3 Solution
   - Recursive construction
   - XSLT implementation

4 Results and Conclusion
   - An example
   - Conclusion
Transformation rules found in the literature are for scalar valued variables

- Consider $z = e(x)$
- Adjoint statement: $\bar{x} += \frac{\partial e}{\partial x} \bar{z}$
- What happens when $e(x) = a \ast x \ast b$ and the variables are matrices?
- What is $\frac{\partial e}{\partial x}$ in this case?
- Dimensions of $\frac{\partial e}{\partial x}$ and $\bar{z}$ will not fit in general.
Outline

1 Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2 Analysis
   - Adjoint rules
     - Example statement
     - Intermediate canonicalization

3 Solution
   - Recursive construction
   - XSLT implementation

4 Results and Conclusion
   - An example
   - Conclusion
Adjoint rules: Sine

- Sine: \( Z = \sin(A) \)

- Derivative variables have the same dimension as the variable they are associated with.

- Use upper case letters to indicate rule place holders.

- The place where \( dZ = \bar{Z} \) occurs is the adjoint position.

\[
\begin{align*}
Z &= \sin(A) \\
A &= \bar{A} + \bar{Z} \cdot \cos(A) \\
dA \cdot \cos A &= dZ \cdot \bar{A}
\end{align*}
\]
Adjoint rules: Multiplication

- Multiplication
  - \( Z = A \times B \), w.r.t. \( A \)
    - \( \overline{A} += \overline{Z} \times B^T \)
    - \( d\overline{A} \times \overline{dZ} ^ T \)
    - \( dA \times \overline{dZ} \)
  - \( Z = A \times B \), w.r.t. \( B \)
    - \( \overline{B} += A^T \times \overline{Z} \)
    - \( dB \times \overline{dZ} \)
    - \( T \times \overline{dZ} \)
1 Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2 Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3 Solution
   - Recursive construction
   - XSLT implementation

4 Results and Conclusion
   - An example
   - Conclusion
Example statement

Consider the statement: \( z = \sin(a \times b) \times c \)

Syntax tree

```
        =
       /\    
      /  \   
     z *   
    /     
   /      
 sin     c
 / \
/  \
/    
/     
/      
/       
a * b
```
Outline

1. Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2. Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3. Solution
   - Recursive construction
   - XSLT implementation

4. Results and Conclusion
   - An example
   - Conclusion
Canonicalization: $s = a \times b$, $t = \sin(s)$, $z = t \times c$
Adjoint canonicalized statements

- Reverse adjoint statements:
  \[
  \bar{t} = \bar{z} \ast c^T, \quad \bar{s} = \cos(s) \ast \bar{t}, \quad \bar{b} += a^T \ast \bar{s}
  \]

Syntax trees:
Putting it back together

- Single adjoint statement: \( \overline{b} + = a^T \ast (\cos(a \ast b) \ast (\overline{z} \ast c^T)) \)

Merged syntax tree

```
+ =
  \overline{db}
  *
    T
    .*
      a
      cos
      *
        s
        dz
        T
        c
```
Outline

1 Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2 Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3 Solution
   - Recursive construction
   - XSLT implementation

4 Results and Conclusion
   - An example
   - Conclusion
Observations

- The outermost \( \ast \)-operator has the adjoint of the statement LHS at the adjoint position.
- The second level operator \( \sin \) has at the adjoint position the expression sub-tree that was produced by the outermost operator.
- The evaluation order is inverted.
Consider a statement $z = e$, assigning expression $e(x)$ to $z$. What is the adjoint expression w.r.t. $x$ for that statement.

Let $a_1 = \text{adjexp}(e, a_0)$ be the adjoint expression of the outermost operation in expression $e$, where $a_0 = \overline{z}$.

Then, $a_2 = \text{adjexp}(\text{active-child}(e), a_1)$ is the adjoint expression of the second level operator, where active-child selects the child of the top along the path to variable $x$.

Finally, $a_k$ is the adjoint of the expression $e$ w.r.t. variable $x$, where $k$ is the depth of $x$ in $e$. 
Outline

1 Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2 Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3 Solution
   - Recursive construction
   - XSLT implementation

4 Results and Conclusion
   - An example
   - Conclusion
Recursive template calls

- Traverse the expression tree in top down order with apply-template
- Pass along two parameters
  - wrt – id of variable node in the expression
  - adj – adjoint constructed so far
- `xsl:apply-templates` to that child which has the node with id `$wrt` among its descendant –or– `self::.`
  - wrt – `$wrt`
  - adj – new expression according to rule for current node, inserting `$adj` at the adjoint position of the rule
Template For Sine

```xml
<xsl:template match="call[id_.="sin"]" mode="diff">
  <xsl:param name="wrt"/>
  <xsl:param name="adj"/>
  <xsl:apply-templates select="*[2]/*" mode="diff">
    <xsl:with-param name="wrt" select="$wrt"/>
    <xsl:with-param name="adj"/>
      <binary op=".*">
        <call>
          <id>cos</id>
          <call>
            <xsl:apply-templates select="*[2]"/>
          </call>
          <xsl:copy-of select="$adj"/>
        </binary>
      </xsl:with-param>
    </xsl:apply-templates>
  </xsl:template>
```

Johannes Willkomm
Adjoint expressions for Matlab
Template For Multiplication

```
<xsl:template match="binary [@op='*']" mode="diff">
    <xsl:param name="wrt"/>
    <xsl:param name="adj"/>
    <xsl:variable name="which" select="*[descendant-or-self::*[generate-id(.)=descendant-or-self::*]]"/>
    <xsl:apply-templates select="$which" mode="diff">
        <xsl:with-param name="wrt" select="$wrt"/>
        <xsl:with-param name="adj"/>
        <xsl:choose>
            <xsl:when test="count($which/following-sibling::*)>1"
                <adjoint-left-multiplication>
                    <xsl:copy-of select="$which"/>
                    <xsl:copy-of select="$adj"/>
                    <xsl:copy-of select="*[2]"/>
                </adjoint-left-multiplication>
            </xsl:when>
            <xsl:otherwise>
            <!-- ... -->
        </xsl:otherwise>
        </xsl:choose>
    </xsl:with-param>
</xsl:apply-templates>
</xsl:template>
```
Template For Variable Nodes

```xml
<xsl:template match="var" mode="diff">
  <xsl:param name="wrt"/>
  <xsl:param name="adj"/>
  <xsl:choose>
    <xsl:when test="generate-id() = $wrt">
      <xsl:copy-of select="$adj"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:literal>0</xsl:literal>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```
Template For Statement

```xml
<xsl:template match="binary [@op = ' = ']
  mode="adjoint">
  <!— pop variables, ... —->
  <xsl:apply-templates
    mode="adjoint-assignment">
    select="*[2]/descendant-or-self::var">
    <xsl:with-param
      name="this" select="."/>
  </xsl:apply-templates>
  <!— zero adjoint of variable written, ... —->
</xsl:template>

<xsl:template match="var"
  mode="adjoint-assignment">
  <xsl:param
    name="this"/>
  <xsl:param
    name="adj">
    <xsl:apply-templates
      select="ancestor::binary[@op = ' = '][1]/*[1]"
      mode="adjoint-var-of-statement"/>
  </xsl:param>
  <xsl:variable
    name="myid"
    select="generate-id()"/>
  <adjoint-increment>
    <target>
      <xsl:apply-templates
        select="(parent::array | .)[1]"/>
    </target>
    <incr>
      <xsl:apply-templates
        select="$this/*[2]" mode="diff">
        <xsl:with-param name="wrt" select="generate-id(.)"/>
        <xsl:with-param name="adj" select="$adj"/>
      </xsl:apply-templates>
    </incr>
  </adjoint-increment>
</xsl:template>
```

Johannes Willkomm

Adjoint expressions for Matlab
Outline

1. Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2. Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3. Solution
   - Recursive construction
   - XSLT implementation

4. Results and Conclusion
   - An example
   - Conclusion
Example: A Times B Times C

```matlab
function z = mult3(a, b, c)
    z = a * b * c;
end
```

```matlab
function [a_a, a_b, a_c, nr_z] = a_mult3(a, b, c, a_z)
    z = a * b * c;
    nr_z = z;
    [a_a a_b a_c] = a_zeros(a, b, c);
    a_a = a_a + a_z*(b * c).';
    a_b = a_b + a.'*a_z*c.';
    a_c = a_c + b.'*a.'*a_z;
end
```

admproc -s adjoint-reductions='no' --nocanonicalize mult3.m
Outline

1 Motivation
   - Generating adjoint code for Matlab
   - Scalar adjoint rules are not enough

2 Analysis
   - Adjoint rules
   - Example statement
   - Intermediate canonicalization

3 Solution
   - Recursive construction
   - XSLT implementation

4 Results and Conclusion
   - An example
   - Conclusion
Conclusion

- Adjoint expressions for arbitrary nested expressions
- Ability to turn off code canonicalization
- Ability to generate code for only scalars or only matrices
- Simple implementation in XSLT
Outlook

- Represent derivative rules in one format for both forward and reverse?
- Handle cases where adjoints are given by algorithm, not an expression