Apache

Commons Nabla

Phil Steitz

Apachecon US, 2011

ACON

Leading the Wave
of Open Source

Agenda

- Motivation - why differentiate bytecode?
- Current API
- How Nabla works internally
- Getting involved - welcome to Commons!

Motivation

Who needs derivatives?

- Lots of numerical algorithms use derivatives or gradients directly
- Derivatives can be useful in diagnostics:
  - Bounding errors
  - Estimating the impacts of parameter or model changes
- Forecast or describe the local behavior of functions

Motivation

But painful and error-prone to compute...

- Two choices:
  - Code analytical derivatives (differentiate functions and code derivative functions manually)
  - Use numerical differentiation techniques
- Manual coding takes time, is error prone, and can be difficult when functions have complex definitions involving loops and conditionals
- Numerical differentiation is not exact, requires configuration settings that can be tricky to determine for complex functions, and has to be done piecewise for functions with conditional branching

Motivation

What if it were possible to get exact derivatives generated automatically by analyzing and engineering the byte code of source functions?

- No need to manually code derivatives
- No need to select algorithms or choose configuration parameters for numerical differentiation
- No need to deal with piecewise definitions, singularities separately
- No separate compilation or code generation

Nabla API

Classes including differentiable functions implement **UnivariateDifferentiable** interface

UnivariateDifferentiator implementations provided by Nabla differentiate bytecode

UnivariateDifferentiable classes get inner classes attached, including derivative functions

The derivative of a function `double f(double t)` is a new function `DifferentialPair f(DifferentialPair t)` in the derivative class

Key Abstractions

```
public class DifferentialPair implements Serializable {
 ...
    /** Value part of the differential pair. */
    private final double value;

    /** First derivative part of the differential pair. */
    private final double firstDerivative;

    public DifferentialPair(final double u0, final double u1) {
        this.value = u0;
        this.firstDerivative = u1;
    }
```

Represents a (value, derivative) pair

Key Abstractions

```
public interface UnivariateDerivative {

    /**
     * The original function associated with this differential.
     */
    UnivariateDifferentiable getPrimitive();

    /**
     * The derivative of the original function.
     */
    DifferentialPair f(DifferentialPair t);

}
```

Differentiators produce derivatives bound to the original function via the **primitive**

Derivatives take **DifferentialPairs** as arguments

Example

```
UnivariateDifferentiable function = new UnivariateDifferentiable() {
    public double f(double t) {
        return t * t + 2 * t + 2;
    }
};

UnivariateDifferentiator differentiator =
    new ForwardModeAlgorithmicDifferentiator();
UnivariateDerivative derivative =
    differentiator.differentiate(function);

DifferentialPair t = DifferentialPair.newVariable(1);

derivative.f(t).getFirstDerivative(); // 2*1 + 2 = 4

derivative.f(t).getValue(); // returns f(1) = 5

derivative.f(new DifferentialPair(1, 2)).getFirstDerivative();
// returns "chain rule" result:  2 * 4 = 8
```

Instance Data

```
public class ParametricFunction implements UnivariateDifferentiable {
    ...
    public void setX(double x) {
        this.x = x;
    }
    public double f(double t) {
        return t * t + x * t + 1;
    }
```

```
ParametricFunction function = new ParametricFunction(1);
final UnivariateDerivative derivative =
    new ForwardModeAlgorithmicDifferentiator().differentiate(function);

DifferentialPair t = DifferentialPair.newVariable(1);
derivative.f(t).getFirstDerivative(); // returns 2t + 1 = 3

function.setX(2);
derivative.f(t).getFirstDerivative(); // now returns 4
```

Singularity

```
UnivariateDifferentiable kinkFunction = new UnivariateDifferentiable() {
    public double f(double t) {
        if (t < 10) {
            return t;
        } else {
            return 2 * t;
        }
    }
};
UnivariateDifferentiator differentiator =
    new ForwardModeAlgorithmicDifferentiator();
UnivariateDerivative derivative = differentiator.differentiate(kinkFunction);

derivative.f(DifferentialPair.newVariable(5)).
       getFirstDerivative();  // Returns 1

derivative.f(DifferentialPair.newVariable(10)).
       getFirstDerivative();  // Returns 2
```

Partial Derivatives

```
public class SetableParameterModel implements UnivariateDifferentiable {

        private Model model;
        private double firstParameter;

        public SetableParameterModel(Model model, double firstParameter) {
            this.model = model;
            this.firstParameter = firstParameter;
        }

        public void setFirstParameter(double firstParameter) {
            this.firstParameter = firstParameter;
        }

        public double f(double t) {
            return model.evaluate(firstParameter, t);
        }

    }
```

Derivative will be partial derivative of model with respect to the second variable (for fixed first variable)

DifferentialPair Operations

```
/**
 * Multiplication operator for differential pairs.
 *
 * @param a left hand side parameter of the operator
 * @param b right hand side parameter of the operator
 * @return a&times;b
 */
public static DifferentialPair multiply(final DifferentialPair a,
                                        final DifferentialPair b) {
    return new DifferentialPair(a.value * b.value,
            a.firstDerivative * b.value + a.value * b.firstDerivative);
}
```

Implementation Strategy

Create an inner class containing the derivative function

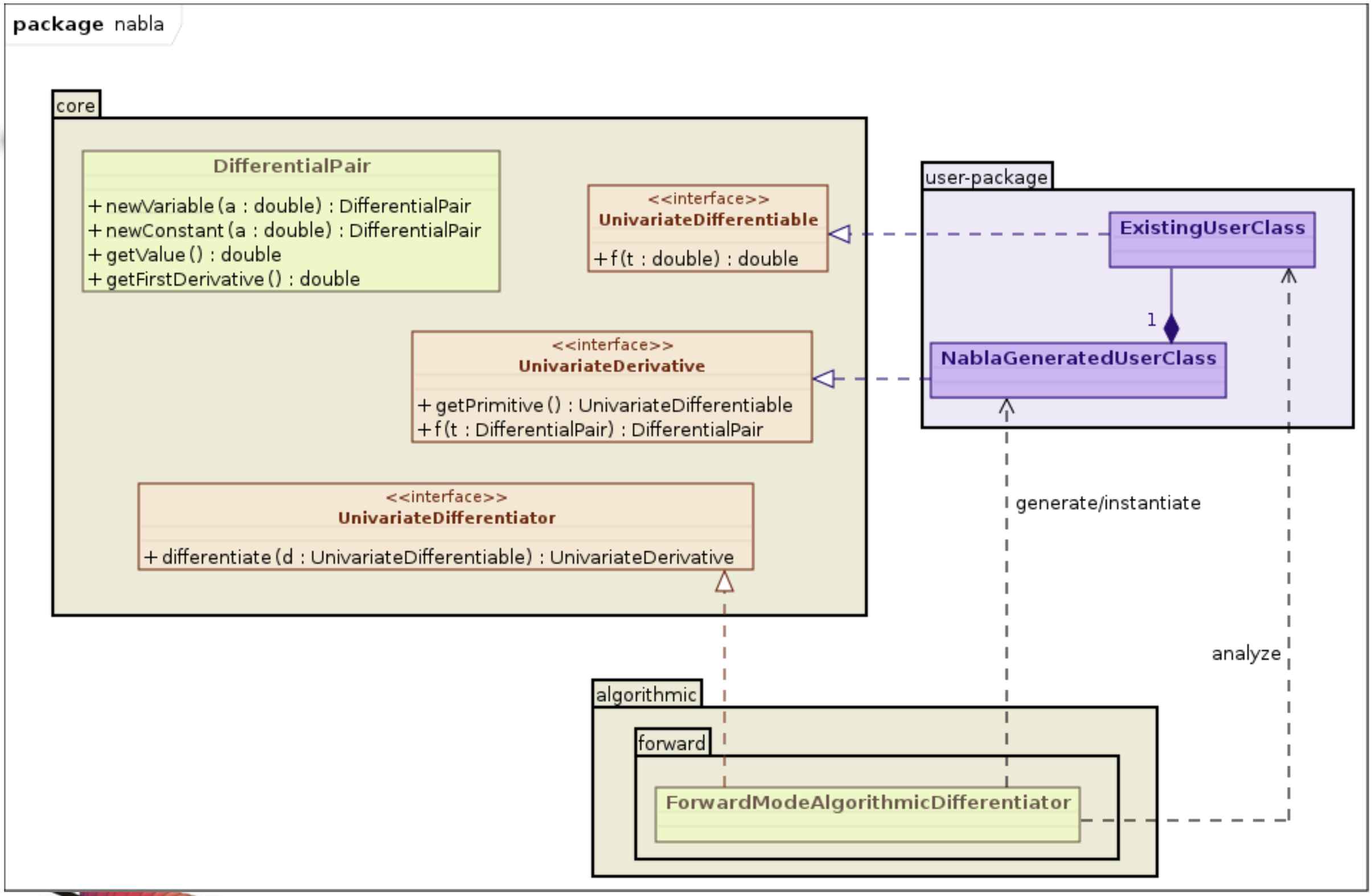
```
public DifferentialPair f(DifferentialPair differentialPair)
```

and a reference to the parent instance

```
public UnivariateDifferentiable getPrimitive()
```

“Surgically attached” derivative instance can refer to instance data of the parent class

- State changes can be reflected in derivatives


package nabla
core
DifferentialPair
+ newVariable (a : double) : DifferentialPair
+ newConstant (a : double) : DifferentialPair
+ getValue () : double
+ getFirstDerivative () : double
<<interface>>
UnivariateDifferentiable
+ f (t : double) : double
<<interface>>
UnivariateDerivative
+ getPrimitive () : UnivariateDifferentiable
+ f (t : DifferentialPair) : DifferentialPair
<<interface>>
UnivariateDifferentiator
+ differentiate (d : UnivariateDifferentiable) : UnivariateDerivative
user-package
ExistingUserClass
1
NablaGeneratedUserClass
generate/instantiate
analyze
algorithmic
forward
ForwardModeAlgorithmicDifferentiator

Implementation Strategy

```
public UnivariateDerivative differentiate(final UnivariateDifferentiable d)
    throws DifferentiationException {

    // get the derivative class
1   final Class<? extends UnivariateDerivative> derivativeClass =
        getDerivativeClass(d.getClass());

    // get the constructor for the child class
2   final Constructor<? extends UnivariateDerivative> constructor =
        derivativeClass.getConstructor(d.getClass());

3   return constructor.newInstance(d);
```

- Transform bytecode to create derivative class
- Include a constructor that takes an instance of the parent
- Create an instance attached to the class being differentiated

Algorithmic Bytecode Differentiation

Analyze the bytecode of f(t), tracing the data flow from t to the final result and differentiate the bytecode of the instructions that consume or produce values along the way.

Look at the instructions that transform the value of t as a composite function:

$$t \rightarrow f_0(t) \rightarrow f_1(f_0(t), t) \rightarrow \ldots \rightarrow f_n(f_{n-1}(\ldots, t), \ldots, t)$$

If we carry along the value of the derivative at t at each stage, we can differentiate the composite sequentially:

$$(t, dt) \rightarrow$$

$$\left(f_0(t)\ ,\ f_0'(t)dt\right) \rightarrow$$

$$\left(f_1(f_0(t), t)\ ,\ f_{11}(f_0(t), t)f_0'(t)dt + f_{12}(t)dt\right) \rightarrow \ldots$$

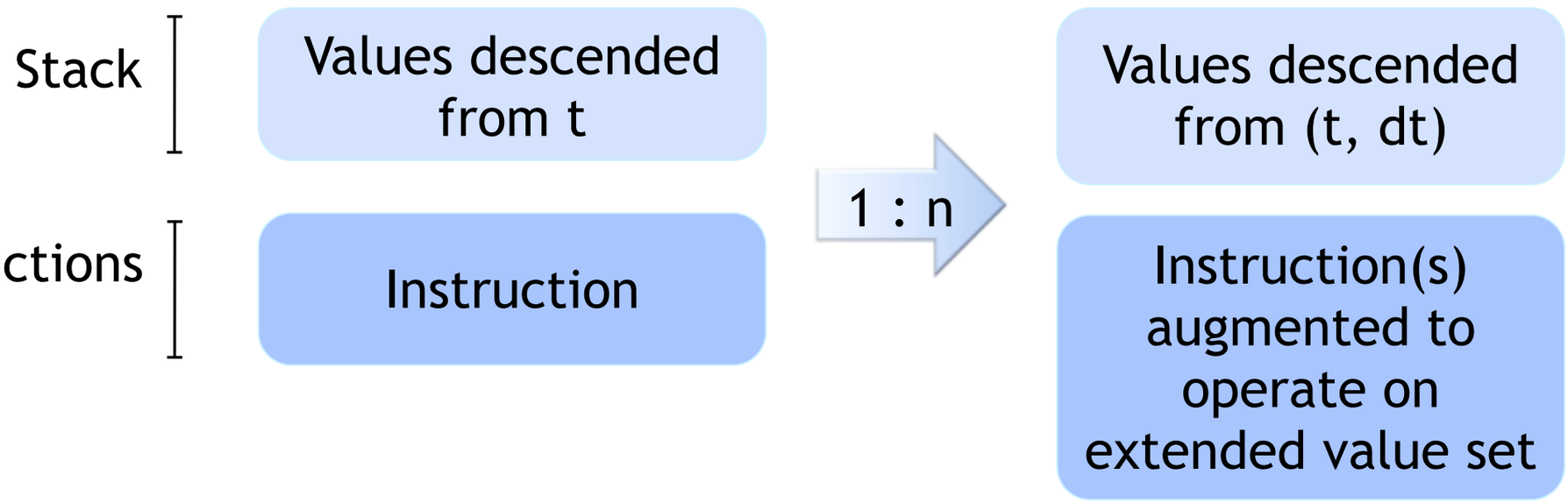
Example

$$t \to t^2 \to t^2 + 2t \to t^2 + 2t + 2$$

$$(t, dt) \to \left(t^2\ ,\ 2tdt\right) \to$$
$$\left(t^2 + 2t\ ,\ 2tdt + 2dt\right) \to$$
$$\left(t^2 + 2t + 2\ ,\ 2tdt + 2dt\right)$$

Transformation Strategy

Identify instructions that use or produce values descended from the actual parameter and transform them to work with DifferentialPairs



Key insight: **this can be done locally - one instruction at a time**

Function bytecode can be ***differentiated as a stream***

Example

Multiplication - **DMUL** instruction operating on t, t (to compute t^2)

- Assume derivative computation starts with **t dt t dt** on the stack
- Replace single DMUL instruction operating on **t t** stack with a sequence that works on **t dt t dt** to yield derivative pair

	Initial Stack	Final Stack
Original (DMUL)	**t t**	**t^2**
Derivative Instructions	**t dt t dt**	**t^2 2tdt**

Does not matter what values start on the stack - just arrange to have **v dv** in place of **v** when derivative sequence starts

Code sequences like this for all basic instructions and built-ins from Java.Math and "*derivative streaming*" just works!

Example

Function to differentiate

```
public double f(double t) {
    return t * t + 2 * t + 2;
}
```

Bytecode	Stack
public f(D)D	
DLOAD 1	t
DLOAD 1	t t
DMUL	t^2
LDC 2.0	t^2 2
DLOAD 1	t^2 2 t
DMUL	t^2 2t
DADD	t^2+2t
LDC 2.0	t^2+2t 2
DADD	t^2+2t+2
DRETURN	
MAXSTACK = 6	
MAXLOCALS = 3	
}	

Example

```
public f(Lorg/.../DifferentialPair;) Lorg/.../DifferentialPair;
    ALOAD 1
    DUP
    INVOKEVIRTUAL org/.../DifferentialPair.getValue ()D
    DSTORE 1
    INVOKEVIRTUAL org/.../DifferentialPair.getFirstDerivative ()D
    DSTORE 3
    DLOAD 1
    DLOAD 3
    DLOAD 1
    DLOAD 3
```

Initial Stack: t dt t dt

Example

```
public f(D)D
    DLOAD 1
    DLOAD 1
    DMUL
    LDC 2.0
    DLOAD 1
    DMUL
    DADD
    LDC 2.0
    DADD
    DRETURN
    MAXSTACK = 6
    MAXLOCALS = 3
}
```

```
                t dt t dt
DSTORE 9        t dt t          dt -> 9
DUP2            t dt t t
DSTORE 7        t dt t          t -> 7
DMUL            t dt*t
DSTORE 5        t               dt*t -> 5
DUP2            t t
DLOAD 9         t t dt
DMUL            t t*dt
DLOAD 5         t t*dt dt*t
DADD            t 2tdt
DSTORE 9        t               2tdt -> 9
DLOAD 7         t t
DMUL            t²
DLOAD 9         t² 2tdt
```

Example

```
public f(D)D
    DLOAD 1
    DLOAD 1
    DMUL
    LDC 2.0
    DLOAD 1
    DMUL
    DADD
    LDC 2.0
    DADD
    DRETURN
    MAXSTACK = 6
    MAXLOCALS = 3
}
```

	t^2 2tdt	
LDC 2.0	t^2 2tdt 2	
DLOAD 1	t^2 2tdt 2 t	
DLOAD 3	t^2 2tdt 2 t dt	
DSTORE 9	t^2 2tdt 2 t	dt->9
DUP2_X2	t^2 2tdt t 2 t	
POP2	t^2 2tdt t 2	
DUP2	t^2 2tdt t 2 2	
DLOAD 9	t^2 2tdt t 2 2 dt	
DMUL	t^2 2tdt t 2 2dt	
DSTORE 9	t^2 2tdt t 2	2dt->9
DMUL	t^2 2tdt 2t	
DLOAD 9	t^2 2tdt 2t 2dt	

Example

```
public f(D)D
    DLOAD 1
    DLOAD 1
    DMUL
    LDC 2.0
    DLOAD 1
    DMUL
    DADD
    LDC 2.0
    DADD
    DRETURN
    MAXSTACK = 6
    MAXLOCALS = 3
}
```

```
                t² 2tdt 2t 2dt
DSTORE 9        t² 2tdt 2t          2dt -> 9
DUP2_X2         t² 2t 2tdt 2t
POP2            t² 2t 2tdt
DLOAD 9         t² 2t 2tdt 2dt
DADD            t² 2t 2tdt+2dt
DSTORE 9        t², 2t              2tdt+2dt -> 9
DADD            t² + 2t
DLOAD 9         t² + 2t, 2tdt + 2dt
```

Example

```
public f(D)D
    DLOAD 1
    DLOAD 1
    DMUL
    LDC 2.0
    DLOAD 1
    DMUL
    DADD
    LDC 2.0
    DADD
    DRETURN
    MAXSTACK = 6
    MAXLOCALS = 3
}
```

	t^2+2t $2tdt+2dt$	
LDC 2.0	t^2+2t $2tdt+2dt$ 2	
DUP2_X2	t^2+2t 2 $2tdt+2dt$ 2	
POP2	t^2+2t 2 $2tdt+2dt$	
DSTORE 9	t^2+2t 2	$2tdt+2dt$ -> 9
DADD	t^2+2t+2	
DLOAD 9	t^2+2t+2 $2tdt+2dt$	

Example

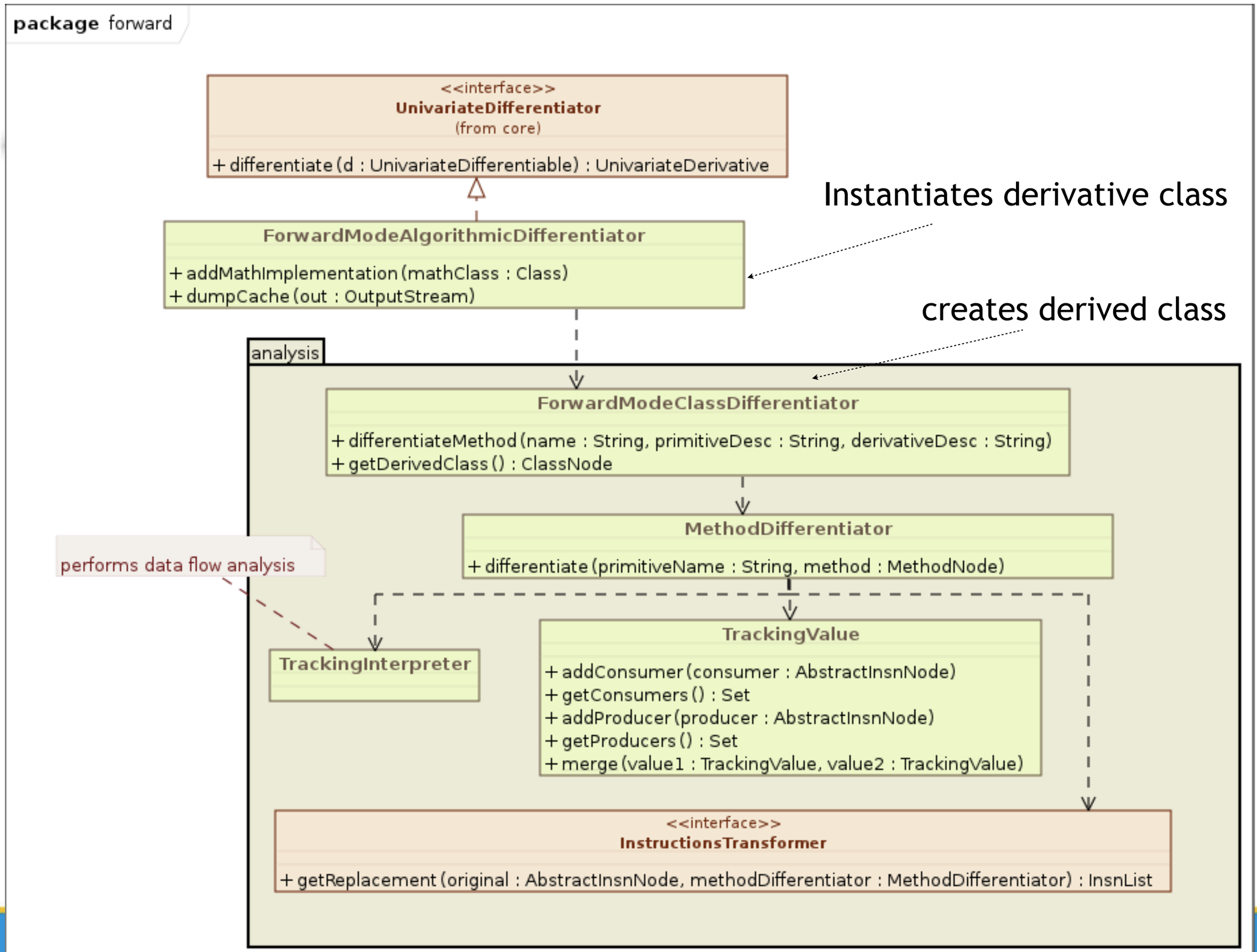
```
public f(D)D
    DLOAD 1
    DLOAD 1
    DMUL
    LDC 2.0
    DLOAD 1
    DMUL
    DADD
    LDC 2.0
    DADD
    DRETURN
    MAXSTACK = 6
    MAXLOCALS = 3
}
```

```
                t²+2t+2 2tdt+2dt
DSTORE 3        t²+2t+2         2tdt+2dt -> 3
DSTORE 1                         t²+2t+2 -> 1
NEW org/.../DifferentialPair
DUP
DLOAD 1         t²+2t+2
DLOAD 3         t²+2t+2 2tdt+2dt
INVOKESPECIAL org/.../DifferentialPair.<init> (DD)V
ARETURN
```

Some Details

Three problems to solve:

1. Creating the derivative class and instantiating it
2. Data flow analysis
3. Differentiating instructions


package forward
<<interface>>
UnivariateDifferentiator
(from core)
+ differentiate (d : UnivariateDifferentiable) : UnivariateDerivative
ForwardModeAlgorithmicDifferentiator
+ addMathImplementation (mathClass : Class)
+ dumpCache (out : OutputStream)
Instantiates derivative class
creates derived class
analysis
ForwardModeClassDifferentiator
+ differentiateMethod (name : String, primitiveDesc : String, derivativeDesc : String)
+ getDerivedClass () : ClassNode
MethodDifferentiator
+ differentiate (primitiveName : String, method : MethodNode)
performs data flow analysis
TrackingInterpreter
TrackingValue
+ addConsumer (consumer : AbstractInsnNode)
+ getConsumers () : Set
+ addProducer (producer : AbstractInsnNode)
+ getProducers () : Set
+ merge (value1 : TrackingValue, value2 : TrackingValue)
<<interface>>
InstructionsTransformer
+ getReplacement (original : AbstractInsnNode, methodDifferentiator : MethodDifferentiator) : InsnList

Implementation Strategy

```
public UnivariateDerivative differentiate(final UnivariateDifferentiable d)
    throws DifferentiationException {

    // get the derivative class
1   final Class<? extends UnivariateDerivative> derivativeClass =
        getDerivativeClass(d.getClass());

    // load the constructor for the child
2   final Constructor<? extends UnivariateDerivative> constructor =
        derivativeClass.getConstructor(d.getClass());

3   return constructor.newInstance(d);
```

- Transform bytecode to create derivative class
- Include a constructor that takes an instance of the parent
- Create an instance attached to the class being differentiated

differentiateMethod in MethodDifferentiator

```
public void differentiate(final String primitiveName, final MethodNode method)
        throws DifferentiationException {
        try {
        ...
        // add spare cells to hold new variables if needed
        addSpareLocalVariables(method.instructions);

        // analyze the original code, tracing values production/consumption
        final FlowAnalyzer analyzer =
            new FlowAnalyzer(new TrackingInterpreter(), method.instructions);
        final Frame[] array = analyzer.analyze(primitiveName, method);
        ...
        // identify the needed changes
        final Set<AbstractInsnNode> changes = identifyChanges(method.instructions);
        ...
        // perform the code changes
        changeCode(method.instructions, changes);
        ...
        // change the method properties to the derivative ones
        method.desc       = Descriptors.DP_RETURN_DP_DESCRIPTOR;
        method.access    |= Opcodes.ACC_SYNTHETIC;
        method.maxLocals  = maxVariables();
```

differentiateMethod in MethodDifferentiator

ASM analyzer does semantic analysis of instructions, revealing symbolic state of stack frame at each instruction execution step

TrackingInterpreter keeps track of which values are produced and consumed by which instructions

```
// analyze the original code, tracing values production/consumption
final FlowAnalyzer analyzer =
    new FlowAnalyzer(new TrackingInterpreter(), method.instructions);
final Frame[] array = analyzer.analyze(primitiveName, method);
```

differentiateMethod in MethodDifferentiator

```
// identify the needed changes
final Set<AbstractInsnNode> changes = identifyChanges(method.instructions);
...
```

Identification of instructions needing change is based on data flow analysis

1. Change the local variables in the initial frame to match the parameters of the derivative method
2. Propagate these variables following the instructions path, updating stack cells and local variables as needed

   ➡ Instructions that must be changed are the ones that consume changed variables or stack cells

changeCode in MethodDifferentiator

```
private void changeCode(final InsnList instructions, final Set<AbstractInsnNode> changes)
        throws DifferentiationException {

    // insert the parameter conversion code at method start
    final InsnList list = new InsnList();
    list.add(new VarInsnNode(Opcodes.ALOAD, 1));
    list.add(new InsnNode(Opcodes.DUP));
    list.add(new MethodInsnNode(Opcodes.INVOKEVIRTUAL, Descriptors.DP_NAME,
                                "getValue", Descriptors.VOID_RETURN_D_DESCRIPTOR));
    list.add(new VarInsnNode(Opcodes.DSTORE, 1));
    list.add(new MethodInsnNode(Opcodes.INVOKEVIRTUAL, Descriptors.DP_NAME,
            "getFirstDerivative", Descriptors.VOID_RETURN_D_DESCRIPTOR));
    list.add(new VarInsnNode(Opcodes.DSTORE, 3));

    instructions.insertBefore(instructions.get(0), list);

    // transform the existing instructions
    for (final AbstractInsnNode insn : changes) {
        instructions.insert(insn, getReplacement(insn));
        instructions.remove(insn);
    }
```

1. Get “t dt” on the stack from the input DifferentialPair
2. Selectively transform instructions needing change “in place”

Example instruction replacement - for DMUL instructions with both arguments DifferentialPairs

```
public InsnList getReplacement(final AbstractInsnNode insn,
                               final MethodDifferentiator methodDifferentiator)
    throws DifferentiationException {
    // Create tmp1-tmp3 temp vars, list as instruction list
    ...
    // operand stack initial state: a0, a1, b0, b1
    list.add(new VarInsnNode(Opcodes.DSTORE, tmp1)); // => a0, a1, b0
    list.add(new InsnNode(Opcodes.DUP2));            // => a0, a1, b0, b0
    list.add(new VarInsnNode(Opcodes.DSTORE, tmp2)); // => a0, a1, b0
    list.add(new InsnNode(Opcodes.DMUL));            // => a0, a1*b0
    list.add(new VarInsnNode(Opcodes.DSTORE, tmp3)); // => a0
    list.add(new InsnNode(Opcodes.DUP2));            // => a0, a0
    list.add(new VarInsnNode(Opcodes.DLOAD,  tmp1)); // => a0, a0, b1
    list.add(new InsnNode(Opcodes.DMUL));            // => a0, a0*b1
    list.add(new VarInsnNode(Opcodes.DLOAD,  tmp3)); // => a0, a0*b1, a1*b0
    list.add(new InsnNode(Opcodes.DADD));            // => a0, a0*b1+a1*b0
    list.add(new VarInsnNode(Opcodes.DSTORE, tmp1)); // => a0
    list.add(new VarInsnNode(Opcodes.DLOAD,  tmp2)); // => a0, b0
    list.add(new InsnNode(Opcodes.DMUL));            // => a0*b0
    list.add(new VarInsnNode(Opcodes.DLOAD,  tmp1)); // => a0*b0, a0*b1+a1*b0

    return list;
```

Work in Progress

- GETFIELD does not really work yet
  - Despite being an inner class, we don't have access out of the box (hidden accessors not created for us)
- INVOKEVIRTUAL does not work at all yet
  - Vision is to maintain a stack and differentiate all relevant methods of any class involved
- Direct support for partial derivatives and gradients
- Reverse mode differentiation

Get Involved!

1. Subscribe to Commons-dev

   http://commons.apache.org/mail-lists.html
2. Check out the code

   http://commons.apache.org/svninfo.html
3. Review open issues

   http://commons.apache.org/sandbox/issue-tracking.html
4. Talk about your ideas
5. Attach patches to JIRA tickets

   http://commons.apache.org/patches.html
6. THANKS!!