Efficient Implementation of a Higher-Order Language with Built-In AD

Jeffrey Mark Siskind, qobi@purdue.edu
Barak Avrum Pearlmutter, barak@pearlmutter.net

PURDUE UNIVERSITY

AD2016, Tuesday 13 September 2016

Key Idea

Migrate reflective source-to-source transformation
from run time to compile time
with abstract interpretation

Traditional AD by Source-to-Source Transformation

Preprocessor at Compile Time

```
function g(x)
   return x+1
end


function f(x)
   return 2*g(x)
end



... derivative(f, 3) ...
```

Traditional AD by Source-to-Source Transformation

Preprocessor at Compile Time

```
function g(x)
   return x+1
end


function f(x)
   return 2*g(x)
end


local y, y_tangent = f_forward(3, 1)
... y_tangent ...
```

Traditional AD by Source-to-Source Transformation

Preprocessor at Compile Time

```
function g(x)
   return x+1
end


function f_forward(x, x_tangent)
   local y, y_tangent = g_forward(x, x_tangent)
   return return 2*y, 2*y_tangent
end

local y, y_tangent = f_forward(3, 1)
... y_tangent ...
```

Traditional AD by Source-to-Source Transformation

Preprocessor at Compile Time

```
function g_forward(x, x_tangent)
   local y, y_tangent = x, x_tangent
   return x+1, x_tangent
end

function f_forward(x, x_tangent)
   local y, y_tangent = g_forward(x, x_tangent)
   return return 2*y, 2*y_tangent
end

local y, y_tangent = f_forward(3, 1)
... y_tangent ...
```

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end
```

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f)
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"
```

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end")
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end")
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward

called_by(f)
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                        local y, y_tangent = g_forward(x, x_tangent)
                        return return 2*y, 2*y_tangent
                     end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward

called_by(f) ==> {g}
```

--

Source-to-Source Transformation at Run Time

Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward

called_by(f) ==> {g}

function derivative(f, x)
   for g in called_by(f) do compile(transform(code(g))) end
   local y, y_tangent = compile(transform(code(f)))(x, 1)
   return y_tangent
end
--
```

But How Can We Make This Efficient?

```
while not converged() do
   x = x-eta*derivative(f, x)
end
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... add(x, y) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add(x, y) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)
   if DOUBLE=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)
   if false then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)



      return scalar_add(x, y)

end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)



      return scalar_add(x, y)

end

local x = 3, y = 4
... scalar_add(x, y) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)



      return scalar_add(x, y)

end

local x = 3, y = 4
... x+y ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add(x, y) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add(ARRAY, ARRAY) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)
   if ARRAY=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)
   if true then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)

      return vector_add(x, y)



end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... vector_add(x, y) ...
```

A Single Powerful Optimization

```
{x = e1, y = e2}.x
```

A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation

A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation
- can eliminate storage reclamation

A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation
- can eliminate storage reclamation
- can eliminate storage writes

A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation
- can eliminate storage reclamation
- can eliminate storage writes
- can eliminate storage reads

A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation
- can eliminate storage reclamation
- can eliminate storage writes
- can eliminate storage reads
- can eliminate dead code

The Kind of Code People Write in Dynamic Languages

```
function map(f, x)
   y = torch.Tensor(x:size(1))
   for i = 1, x:size(1) do
      y[i] = f(x[i])
   end
   return y
end

function reduce(g, i, x)
   y = i
   for i = 1, x:size(1) do
      y = g(y, x[i])
   end
   return y
end

reduce(function(x, y) return x+y end,
       0,
       map(function(x) return x*x end, torch.Tensor({u, v, w, x, y})))
```

--

The Kind of Code People Write in Dynamic Languages

```
function map(f, x)
   y = torch.Tensor(x:size(1))
   for i = 1, x:size(1) do
      y[i] = f(x[i])
   end
   return y
end

function reduce(g, i, x)
   y = i
   for i = 1, x:size(1) do
      y = g(y, x[i])
   end
   return y
end

reduce(function(x, y) return x+y end,
       0,
       map(function(x) return x*x end, torch.Tensor({u, v, w, x, y})))

u*u + v*v + w*w + x*x + y*y

--
```

Key Idea

You need this anyway
to compile dynamic languages efficiently

Same mechanism can support AD

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative(g, x)
   local y, y_tangent = compile(transform(code(g)))(x, 1)

   return y_tangent
end
```

```
... derivative(f, 3) ...
```

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(g, x)
   local y, y_tangent = compile(transform(code(g)))(x, 1)

   return y_tangent
end
```

```
... derivative_1(FUNCTION_F, 3) ...
```

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(FUNCTION_F, x)
   local y, y_tangent = compile(transform(code(FUNCTION_F)))(x, 1)

   return y_tangent
end

... derivative_1(FUNCTION_F, 3) ...
```

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(FUNCTION_F, x)
   local y, y_tangent = compile(transform("function f(x)
                                              return 2*x
                                           end"))(x, 1)

   return y_tangent
end
```

```
... derivative_1(FUNCTION_F, 3) ...
```

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(FUNCTION_F, x)
   local y, y_tangent = compile("function f_forward(x, x_tangent)
                                    local y, y_tangent = 2*x, 2*x_tangent
                                    return y, y_tangent
                                 end")(x, 1)
   return y_tangent
end
```

```
... derivative_1(FUNCTION_F, 3) ...
```

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end

function f_forward(x, x_tangent)
   local y, y_tangent = 2*x, 2*x_tangent
   return y, y_tangent
end

function derivative_1(FUNCTION_F, x)
   local y, y_tangent = f_forward(x, 1)



   return y_tangent
end


... derivative_1(FUNCTION_F, 3) ...
```

Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end

function f_forward(x, x_tangent)
   local y, y_tangent = 2*x, 2*x_tangent
   return y, y_tangent
end

function derivative(g, x)
   local y, y_tangent = compile(transform(code(g)))(x, 1)



   return y_tangent
end

local y, y_tangent = f_forward(x, 1)
... y_tangent ...
```

A Single Powerful Optimization

A Single Powerful Optimization

- separates AD from optimization

A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms

A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)

A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)

A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)
- many AD optimizations (like TBR) fall out

A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)
- many AD optimizations (like TBR) fall out
- makes it easier to get it right

A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)
- many AD optimizations (like TBR) fall out
- makes it easier to get it right
- makes it easier to get it to nest

Essence of Forward Transform

$$
\begin{array}{rcl}
\overrightarrow{c} & \rightsquigarrow & \overrightarrow{\mathcal{J}}\ c \\
\overrightarrow{\lambda x.e} & \rightsquigarrow & \lambda\,\overrightarrow{x}\,.\,\overrightarrow{e} \\
\overrightarrow{e_1\ e_2} & \rightsquigarrow & \overrightarrow{e_1}\ \overrightarrow{e_2} \\
\overrightarrow{\textbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ e} & \rightsquigarrow & \textbf{letrec}\ \overrightarrow{x_1} = \overrightarrow{e_1}; \ldots; \overrightarrow{x_n} = \overrightarrow{e_n}\ \textbf{in}\ \overrightarrow{e} \\
\overrightarrow{e_1\, e_2} & \rightsquigarrow & \overrightarrow{e_1}\ \overrightarrow{,}\ \overrightarrow{e_2}
\end{array}
$$

Essence of Reverse Transform

$$
\begin{array}{rcl}
\overleftarrow{x = c} & \rightsquigarrow & \overleftarrow{x} = \overleftarrow{\mathcal{J}}\, c \\
\overleftarrow{x_1 = x_2} & \rightsquigarrow & \overleftarrow{x_1} = \overleftarrow{x_2} \\
\overleftarrow{x = \lambda x.e} & \rightsquigarrow & \overleftarrow{x} = \lambda x.e \\
\overleftarrow{x = x_1\ x_2} & \rightsquigarrow & \overleftarrow{x}, \bar{x} = \overleftarrow{x_1}\ \overleftarrow{x_2} \\
\overleftarrow{x = x_1, x_2} & \rightsquigarrow & \overleftarrow{x} = \overleftarrow{x_1}, \overleftarrow{x_2}
\end{array}
$$

$$
\begin{array}{rcl}
\overline{x_1 = x_2} & \rightsquigarrow & \overleftarrow{x_2} \mathrel{+}= \overleftarrow{x_1} \\
\overline{x = \lambda x.e} & \rightsquigarrow & \overleftarrow{\lambda x.e} \mathrel{+}= \overleftarrow{x} \\
\overline{x = x_1\ x_2} & \rightsquigarrow & \overleftarrow{x_1}, \overleftarrow{x_2} \mathrel{+}= \bar{x}\ \overleftarrow{x} \\
\overline{x = x_1, x_2} & \rightsquigarrow & \overleftarrow{x_1}, \overleftarrow{x_2} \mathrel{+}= \overleftarrow{x}
\end{array}
$$

$$
\overleftarrow{\lambda x.\mathbf{let}\ b_1; \ldots; b_n \mathbf{in}\ y} \quad \rightsquigarrow \quad \lambda \overleftarrow{x}.\mathbf{let}\ \overleftarrow{b_1}; \ldots; \overleftarrow{b_n} \mathbf{in}\ \overleftarrow{y}, \lambda \overleftarrow{y}.\mathbf{let}\ \overline{b_n}; \ldots; \overline{b_1} \mathbf{in}\ \overleftarrow{x}
$$

Benchmarks

		backprop		
		Fs	Fv	R
VLAD	STALIN∇	1.00	■	1.00
FORTRAN	ADIFOR	15.51	3.35	■
	TAPENADE	14.97	5.97	6.86
C	ADIC	22.75	5.61	■
C++	ADOL–C	12.16	5.79	32.77
	CPPAD	54.74	■	29.24
	FADBAD++	132.31	46.01	60.71
ML	MLTON	95.20	■	39.90
	OCAML	202.01	■	156.93
	SML/NJ	181.93	■	102.89
HASKELL	GHC	■	■	■
SCHEME	BIGLOO	743.26	■	360.07
	CHICKEN	1626.73	■	1125.24
	GAMBIT	671.54	■	379.63
	IKARUS	279.59	■	165.16
	LARCENY	1203.34	■	511.54
	MIT SCHEME	2446.33	■	1113.09
	MZC	1318.60	■	754.47
	MZSCHEME	1364.14	■	772.10
	SCHEME->C	597.67	■	280.93
	SCMUTILS	5889.26	■	■
	STALIN	435.82	■	281.27

Damned Benchmarks

		particle				saddle			
		FF	FR	RF	RR	FF	FR	RF	RR
VLAD	STALIN∇	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FORTRAN	ADIFOR	2.05	■	■	■	5.44	■	■	■
	TAPENADE	5.51	■	■	■	8.09	■	■	■
C	ADIC	■	■	■	■	■	■	■	■
C++	ADOL–C	■	■	■	■	■	■	■	■
	CPPAD	■	■	■	■	■	■	■	■
	FADBAD++	93.32	■	■	■	60.67	■	■	■
ML	MLTON	78.13	111.27	45.95	32.57	114.07	146.28	12.27	10.58
	OCAML	217.03	415.64	352.06	261.38	291.26	407.67	42.39	50.21
	SML/NJ	153.01	226.84	270.63	192.13	271.84	299.76	25.66	23.89
HASKELL	GHC	209.44	■	■	■	247.57	■	■	■
SCHEME	BIGLOO	627.78	855.70	275.63	187.39	1004.85	1076.73	105.24	89.23
	CHICKEN	1453.06	2501.07	821.37	1360.00	2276.69	2964.02	225.73	252.87
	GAMBIT	578.94	879.39	356.47	260.98	958.73	1112.70	89.99	89.23
	IKARUS	266.54	386.21	158.63	116.85	424.75	527.57	41.27	42.34
	LARCENY	964.18	1308.68	360.68	272.96	1565.53	1508.39	126.44	112.82
	MIT SCHEME	2025.23	3074.30	790.99	609.63	3501.21	3896.88	315.17	295.67
	MZC	1243.08	1944.00	740.31	557.45	2135.92	2434.05	194.49	187.53
	MZSCHEME	1309.82	1926.77	712.97	555.28	2371.35	2690.64	224.61	219.29
	SCHEME->C	582.20	743.00	270.83	208.38	910.19	913.66	82.93	69.87
	SCMUTILS	4462.83	■	■	■	7651.69	■	■	■
	STALIN	364.08	547.73	399.39	295.00	543.68	690.64	63.96	52.93

Statistics

		probabilistic-lambda-calculus		probabilistic-prolog	
		F	R	F	R
VLAD	STALIN∇	1.00	1.00	1.00	1.00
FORTRAN	ADIFOR	■	■	■	■
	TAPENADE	■	■	■	■
C	ADIC	■	■	■	■
C++	ADOL–C	■	■	■	■
	CPPAD	■	■	■	■
	FADBAD++	■	■	■	■
ML	MLTON	129.11	114.88	848.45	507.21
	OCAML	249.40	499.43	1260.83	1542.47
	SML/NJ	234.62	258.53	2505.59	1501.17
HASKELL	GHC	■	■	■	■
SCHEME	BIGLOO	983.12	1016.50	12832.92	7918.21
	CHICKEN	2324.54	3040.44	44891.04	24634.44
	GAMBIT	1033.46	1107.26	26077.48	14262.70
	IKARUS	497.48	517.89	8474.57	4845.10
	LARCENY	1658.27	1606.44	25411.62	14386.61
	MIT SCHEME	4130.88	3817.57	87772.39	49814.12
	MZC	2294.93	2346.13	57472.76	31784.38
	MZSCHEME	2721.35	2625.21	60269.37	33135.06
	SCHEME->C	811.37	803.22	10605.32	5935.56
	SCMUTILS	7699.14	■	83656.17	■
	STALIN	956.47	1994.44	15048.42	16939.28

Take-Home Message

Powerful and efficient AD can be attained by:

- integrating AD into compiler
- formulating AD as one of many compiler transformations
- using abstract interpretation to migrate AD transformation from run time to compile time