Block Scope Differentiation

AD2016 Oxford

Dominic Jones

dominic.jones@gmx.co.uk

London, September 2016

Aim

An additional helping hand

- The aim is to have the fastest possible differentiation tool
- In C++, with 2011 standard available
- Which can integrate with existing framework
- Does not introduce unusual syntax
- A small tool which is easy to use selectively

Target

- Modern, commercial, CFD software
- Adjoint differentation of coupled Navier-Stokes solver
- Calculation of element cell terms via face-cell loops

```
for (f = 0; f != N; ++f)
{
  // cache input fields
  const Vector<3,double> A_f{A[f]};
  const Vector<3,float> U_f{U[f]};
  const double rho_f{rho[f]};

  // compute local values
  const double spd{dot(U_f, A_f)};
  const float flux{rho_f * spd};

  // write results
  R[f](0) += flux * U_f;
  R[f](1) -= flux * U_f;
}
```

Method

l-value destructor

- If the loop block does not have nested scopes then the reverse sequence of destructor operations could be harnessed

```
mode = ADJOINT;
{
  ...

  // compute local values
  const Drv<mode,double> spd{dot(U_f, A_f)};
  const Drv<mode,float> flux{rho_f * spd};

  ...

  ~flux() { rho_f.adj() += spd.pri() * flux.adj();
            spd.adj() += rho_f.pri() * flux.adj(); }

  ~spd() { U_f.adj() += A_f.pri() * spd.adj();
           A_f.adj() += U_f.pri() * spd.adj(); }
}
```

Caching the expression

- With a reasonable guess of the expression size, a copy could be made during construction and then evaluated later by some engine

```
{
  ...

  const Drv<mode,float,expr_size> flux{rho_f * spd};

  ...

  ~flux()
  {
    (*this->_adjointExpression)(this->_expr, this->_adj);
  }
}
```

Typelessness

- At construction, the expression gets copied to a local array and a function pointer to the adjoint expression engine is stored

```
template<Mode m, typename T, int s>
class Drv<m,ExprCache<T,s> > : public Drv<m,T>
{
  using AdjointExpression_t =
    void(*)(void const * const, T const &);

  // constructor
  template<typename Expr_t> Drv(Expr_t const &expr)
    : Drv<m,T>(primalExpression<Expr_t>(expr))
    , _expr(memcpy(sizeof(expr), expr))
    , _adjointExpression(&adjointExpression<Expr_t,T>)
  {}

  // members
  std::array<char,s> _expr;
  AdjointExpression_t _adjointExpression;
}
```

Type retention via auto

- To facilitate auto, an assign function is required to build the correct *l-value* type

```
{
  ...

  // compute local values
  const auto flux{assign(rho_f * spd)};

  ...

  ~flux()
  {
    adjointExpression<Expr_t,T>(this->_expr, this->_adj);
  }
}
```

Expression context

- Separate the context of how an expression is to be evaluated from the expression itself

- Made possible with user-defined types and operator overloading

- But needs to support arithmetic with mixed types (`float`, `double`, `Vector<N,T>`, `Tensor<N,T>`)

- Must co-exist with other operator overloading tools (`PETE`, Los Alamos National Laboratory)

Expression tree

- Nodes templated on operator, result and argument types
- Sub-nodes may be passive (non-differentiable) or active
- Sub-nodes may be owned by value or by reference

```
// expression
dot(U_f, A_f);

// expression type
Binary<Dot,
       double,
       Drv<mode,Vector<3,float>>,
       Drv<mode,Vector<3,double>>
      >
```

Writing to the inputs

- A differentiatable type retains the address either to its own adjoint value else to the adjoint value it was constructed with

```
template<Mode m, typename T>
class Drv : public ExpressionNode<m,T,Drv<T> >
{
  Drv(T const &pri, T &drv)
    : _pri(pri), _drv(drv), _adj(drv)
  {}

  T const &pri() const { return _pri; }
  void adj(T const &rhs) const { _adj += rhs; }

  T _pri, _drv;
  T &_adj;
};
```

Function signature

- Mutable and immutable types have slightly different syntax

```
template<Mode mode>
void
evaluate(const Drv<mode,double> & A,
         const Drv<mode,double> & B,
         Drv<mode,double&> Z)
{
  const Drv<mode,double,10> T0{A * B};
  const Drv<mode,double,15> T1{reciprocal(A + B)};
  Z = T0 * T1;
}
```

Function invocation

- Adjoint evaluation is no more than the function call

```
double a_pri{3}, b_pri{4};

double a_adj{0}, b_adj{0};
double z_adj{1};

evaluate(Drv<mode,double>{a_pri, a_adj},
         Drv<mode,double>{b_pri, b_adj},
         Drv<mode,double&>{z_adj});

std::cout << a_adj << std::end;
std::cout << b_adj << std::end;
```

Testing

Harmonic function

- Test case used by NAG
- 5 inputs, 1 output, 100 lines
- `github.com/DominicJones/AD2016_Oxford`
- Five approaches to evaluating the adjoint:
  1. Adept AD operator overloading tool (13.3x)
  2. Tapenade AD source transformation tool (1.9x)
  3. typeless expression caching (5.8x)
  4. typed expression caching (using `auto`) (5.8x)
  5. naive use of `auto` (320x)
- timings are median average of 50 evaluations, 100,000 iterations per evaluation (g++ 5.1 -O3, Intel Xeon E5-2650)

Further work

Removing the expression copying

- Copying every expression so that it can be used in the destructor is the principle hit on performance
- Instead, make the expression nodes perform the adjoint evaluation in *their* destructors
- `auto` must be used in place of an *l-value* type

Supporting `if` blocks

- Already possible, but tree must always own sub-nodes by value
- Within the nested scope, naive use of `auto` is necessary

`auto` return type

- With the 2014 standard, expression types can be returned from functions
- Removes the need to maintain function call back pointers
- But the expression must hold copies of local variables, rather than references