Discrete Adjoint

Another Perspective

Dominic Jones

December 2013

Leveraging the Compiler

General remarks

- Compilers compile one translation unit at a time; to scale with compilation the generation of derivative code should follow the same design
- Compilers optimise intermediate code; generation of the derivative could benefit from this
- Functional programming lends itself to implementing derivatives of functions, to compiler optimisation and component programming

Technology and trends

- Composing a language from domain-specific embedded languages is an approach taken by Laurence Tratt at King's College: instead of a language providing every feature, present a language which provides the tools to extend the language (but not beyond recognition)
- Template expressiveness and compile time function evaluation features are continually being developed
- The D programming language offers a handful of features which, when considered collectively, are very appealing for writing a compile-time embedded derivative code generator, or at the very least minimising the tediuum, code duplication and run-time costs of typical derivative implementations

Roll your own – Improved

D offers the following helping hands:

- `static if`
- `immutable` types
- `scope(exit)`
- compile time function evaluation via `static auto v = ...`
- `mixin template`
- `std.functional.memoize`

Ingredients: scope (exit)

Given some function, have the adjoint statements written in the same order as the primal statements. So if the function is originally:

```
void eval(in Primal a, in Primal b, out Primal c) {
  auto w1 = a * b;
  auto w2 = w1 * (a - b);
  c = w2 / w1;
}
```

then after being differentiated, it may look something like...

Ingredients: scope (exit)

```
void eval(ref Dual a, ref Dual b, ref Dual c) {
  auto w1 = tuple!Dual(a[0] * b[0], 0);
  scope (exit) {
    b[1] += a[0] * w1[1];
    a[1] += b[0] * w1[1];
  }

  auto w2 = tuple!Dual(w1[0] * (a[0] - b[0]), 0);
  scope (exit) {
    w1[1] += (a[0] - b[0]) * w2[1];
    a[1] -= b[0] * w2[1];
    b[1] += a[0] * w2[1];
  }

  c = tuple!Dual(w2[0] / w1[0], 0);
  scope (exit) {
    w2[1] += c[1] / w1[0];
    w1[1] -= w2[0] * c[1] / w1[0]^^2;
  }
}
```

Ingredients: static if

If the primal, tangent and adjoint statement can be written in the same order (using scope-exit), then perhaps a single function can embody the three versions without run-time overhead

```
void eval(Mode, Type)(ref Type a, ref Type b, ref Type c) {
  auto w1 = Type(a * b);
  static if (Mode.stringof == 'Tangent') {
    w1[1] = a[0] * b[1] + a[1] * b[0];
  }
  else if (Mode.stringof == 'Adjoint') {
    scope (exit) {
      b[1] += a[0] * w1[1];
      a[1] += b[0] * w1[1];
    }
  }

  ...
}
```

Ingredients: CTFE and mixin

Using scope-exit and static-if still looks clumsy. One clean up could then be to parse the primal statement, generating its tangent or adjoint and mixing it in to the code

```
void eval(Mode, Type)(ref Type a, ref Type b, ref Type c) {
  mixin(generate!Mode('auto w1 = Type(a * b);'));

  ...
}
```

where `generate(Mode)(statement)` would return (as a string) the adjoint statements and primal statement if the mode is adjoint, tangent statement and primal statement for the tangent mode and the primal statement for only the primal mode.

Ingredients: CTFE and mixin

- If `generate(Mode)(statement)` is used just in a statement-wise way, the implementation of the function would be quite simple, relying on compiler introspection to fill in the type details.
- A more sophisiticated `generate` parser may operate on a whole function or module. In this case the parser would need to be able to parse the language properly.
  `https://github.com/PhilippeSigaud/Pegged` offers such a parser.
- Having code written as strings is hardly ideal.

Ingredients: mixin modules

It's perhaps better to generate the derivative code in a separate file rather than clutter original implementations of functions.

```
module solver;

void solve(in double x, out double f)
{
  ...
}
```

However, this approach may require significant work to be implemented.

```
module solver_derivative;

mixin(generate!Tangent(import("solver.d")));
mixin(generate!Adjoint(import("solver.d")));

unittest {
  solve_tng(x, x_tng, f, f_tng);
  solve_adj(x, x_adj, f, f_adj);
}
```