Expression Templates in Ada

Alexandre Duret-Lutz

EPITA Research and Development Laboratory

`http://www.lrde.epita.fr/`

May 16, 2001

Context

- Olena: a platform for *generic* image processing
  - written in C++
  - using template programming (a la STL)
- Can template programming be adapted to Ada?
  - Ada has strong support for genericity
  - an STL-like library exists: SGL
  - ⇒ let's experiment with expression templates

Purpose of Expression Templates

Compile a simple expression like

```
Matrix<int,100,100> a, b, c;
[...]
a = b * 5 + c;
```

as if it were written into a faster equivalent

```
Matrix<int,100,100> a, b, c;
[...]
for (int i = [...])
  for (int j = [...])
    a(i,j) = b(i,j) * 5 + c(i,j);
```

getting rid of large and costly temporary variables.

How does it work?

```
Matrix<int,100,100> a, b, c;
a = b * 5 + c ;

Sum< Scalar_Product<Matrix<int,100,100>,int> ,
     Matrix<int,100,100> >

a = temp2;

for (i = [...])
  for (j = [...])
    a(i,j) = temp2(i,j);
                 temp1(i,j) + c(i,j)
                        b(i,j) * 5
```

Language features used

C++: implicit template instantiations ☺

⇒ we are going to make explicit instantiations in Ada ☹

C++: unconstrained generic parameters ☹
(e.g. `Sum<T,U>` without constraints on `T` and `U`)

⇒ *signatures* can express constrinats on generic parameters ☺

Signatures = (usually empty) generic packages

```
generic
   type Values is private;
   type Height_Range is range <>;
   type Width_Range is range <>;
   type Array_Type is array (Height_Range, Width_Range) of Values;
package Matrix_Type is end Matrix_Type;

with Matrix_Type;
generic
   with package Matrix_Spec is new Matrix_Type (<>);
   with function Get_Value (At_Row : in Matrix_Spec.Height_Range;
                            At_Col : in Matrix_Spec.Width_Range)
     return Matrix_Spec.Values;
package Matrix_Expression is
   procedure Assign (To : out Matrix_Spec.Array_Type);
end Matrix_Expression;
```

`Assign` performs the loop, calling `Get_Value`.

Building a Matrix Expression (1/2)

We must conform to the `Matrix_Expression` requirements.

```
with Matrix_Expression;
generic
   with package Left_Expr is new Matrix_Expression (<>);
   with package Right_Expr is new Matrix_Expression (<>);
   with function Operator (Left : in Left_Expr.Matrix_Spec.Values;
                           Right : in Right_Expr.Matrix_Spec.Values)
     return Left_Expr.Matrix_Spec.Values;
package Matrix_Operators.Binary is
   function Get_Value (At_Row : in Left_Expr.Matrix_Spec.Height_Range;
                       At_Col : in Left_Expr.Matrix_Spec.Width_Range)
     return Left_Expr.Matrix_Spec.Values;
   pragma Inline (Get_Value);

   package Expr is
      new Matrix_Expression (Left_Expr.Matrix_Spec, Get_Value);
end Matrix_Operators.Binary;
```


Building a Matrix Expression (2/2)

Any expression is a package
⇒ a matrix must be encapsulated in a package.

Usage:

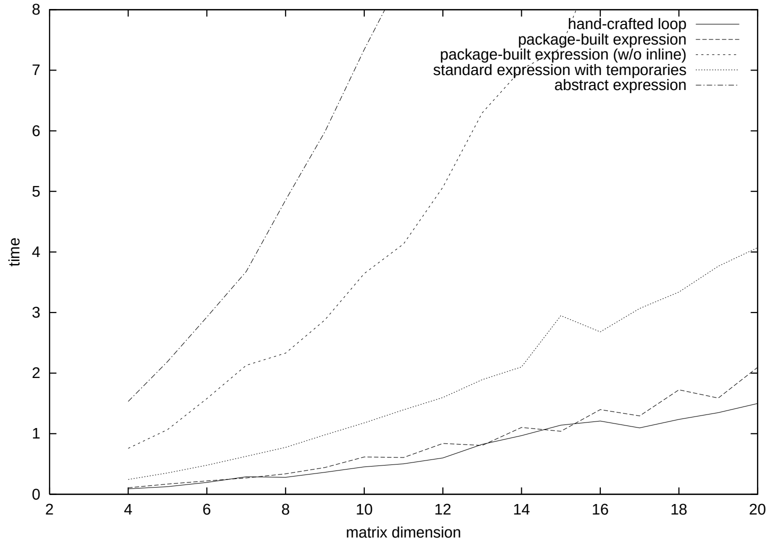
- instantiate a package for each matrix
- instantiate operator packages to build the expression
- `Assign` the result from the final expression

Example (1/2)

```
declare
   type Range3 is range 0 .. 2;
   type Matrix33 is array (Range3, Range3) of Integer;
   package Matrix33_Spec is
      new Matrix_Type (Values => Integer,
                       Height_Range => Range3,
                       Width_Range => Range3,
                       Array_Type => Matrix33);

   B : Matrix33 := ((1, 0, 0), (0, 1, 0), (0, 0, 1));
   C : Matrix33 := ((0, 1, 0), (0, 0, 1), (1, 0, 0));

   package B_Inst is new Matrix_Instance (Matrix33_Spec, B);
   package B_Expr renames B_Inst.Expr;
   package C_Inst is new Matrix_Instance (Matrix33_Spec, C);
   package C_Expr renames C_Inst.Expr;
```


Example (2/2)

```
   -- Build the expression B*5
   package B5_Inst is new Scalar_Binary (B_Expr, 5, "*");
   package B5_Expr renames B5_Inst.Expr;

   -- Build the expression B*5+C
   package Res_Inst is new Binary (B5_Expr, C_Expr, "+");
   package Res_Expr renames Res_Inst.Expr;

   A : Matrix33; -- where the result will be stored
begin
   [...]
   Res_Expr.Assign (To => A);
end;
```

Comments on the result

C++ templates	ADA generics
implicit type instantiation ⇒ easy writting of expressions	*explicit package instantiation* ⇒ lower the usefulness of expression templates
semantic analysis delayed until instantiation ⇒ error-prone	*strict checking* ⇒ helpful, force to express constraints using signatures; yet constraints are not powerful enough

Another use of expression templates

```
img2 = erode (img1);

img2 = erode (mask (img1, msk));

img2 = erode (red_access (mask (img1, msk)));
```


Conclusion

- Expected results as efficiency is concerned
- Too much "administrative overhead"
- Interesting use of signatures, to combine packages
- Yet Ada lacks the ability to express constraints between package parameters

Example source code: `http://www.lrde.epita.fr/download/`