From Mathematica patterns to C++ template metaprogramming

Implementing the Lambda Calculus in one (actually two) bizarre ways

Mathematica is great tool for scientific computation. The C++ templates offer completely new opportunity for creating Turing-complete metaprograms based on the template specialization. I played a lot with both things, they are great, etc., but I've never seen a paper where the great similarity between them is fully revealed. So let's start from the beginning.

The C++ template metaprogramming programs are usually referred as *functional* programs, because of many things, like that you don't have state – you don't have variables, etc. This view of the metaprograms is only one of the possible points of view.

In *Mathematica*, one of the cool features that allow you to write elaborate statements is the pattern-matching technique. You can define a single function to act in different ways when the arguments match some pattern. From now on, I will refer to that paradigm as pattern programming. In fact, the pattern programming itself is Turing-complete and especially useful when it comes to mathematical expressions manipulations. The key idea is, that the way we use patterns in *Mathematica* resembles the way the template metaprograms work – we provide different answers on different specializations (patterns). The analogy is so straight, that we may take an "only-pattern-programming" *Mathematica* program and pretty straightforward translate it in the C++ template metaprogramming language. In fact, it's interesting (at least for me) to get one leap forward – I wanted a *Mathematica* script that takes the *Mathematica* program and produces ready-to-go C++ template code. Indeed, the task is so simple, that implementing the Lambda calculus as template metaprograms requires more work.

There were some difficulties, through, such as; the typical use of patterns in Mathematica is something like this:

```
Fun[a_, b_, Num[c_]] := Pair[b, c];
```

But you cannot safely manipulate such expressions just because the := operator collapses them to Null. So, instead of sequence of SetDelayed-s I chose to identify a program as a list of declarations and simple rules, instead of the SetDelayed. For example, the upper instruction can be carried as:

```
Fun[a_, b_, Num[c_]] → Pair[b, c]
```

This is better, because the production rule doesn't collapse to Null.

Next, in C++, for a template specialization to be used, the template structure must be first declared for the compiler to know how many template arguments has. But the syntax for template declaration and template specialization is different, namely, there is not/is in <> things after the name of the struct, so

they should be treated differently. Anyway, the template definition means nothing to *Mathematica*, so better to be constructed from a single expression. So we need this:

```
IF[c_, t_, e_]
```

to translate into this:

```
template<typename c, typename e, typename t>
struct IF{
static void prt(void){printf("IF[c_, t_, e_]");};
typedef IF<c, t, e>  res;};
```

The prt() function is added to all structs to actually see some result when running the program.

But we want:

```
Eval[Valu[a_], Pair[b_, c_]] → Eval[Valu[a], c]
```

to translate into:

```
template<typename a, typename b, typename c>
struct Eval<Valu<a> , Pair<b, c> > {
static void prt(void){
printf("Eval[Valu[a_], Pair[b_, c_]] -> Eval[Valu[a], c]");};
typedef typename Eval<typename Valu<a>::res , c>::res  res;};
```

Notice that after struct IF there is a { and after struct Eval there is <.

After figured this out, how do you find the pattern placeholders in a *Mathematica* expression? That's why I like it-it has powerful functions that can cope with anything almost without pain:

```
FindAllPatterns[expr_] := (First[#] & /@ Union@Cases[expr, a_Pattern, 1000]);
```

It may look strange to the untrained eye, but all is there! On a single line!

The rest is really boring string manipulation. Here is the Mathematica translator:

```
FindAllPatterns[expr_] := (First[#] & /@ Union@Cases[expr, a_Pattern, 1000]);
Specialisation[expr_] := "template" <> StringReplace[ToString["typename "
        <> (ToString@#) & /@ FindAllPatterns[expr]], {"{" → "<", "}" → ">"}];
SecondRow[expr_] := "struct " <> ToString[Head@expr] <> "{";
SecondRow[expr_ → res_] := "struct " <> StringReplace[ToString[expr],
    {"[" → "<", "]" → "> ", "_" → "", "$" → ""}] <> "{";
PrintInfo[expr_] := "static void prt(void){printf(\"" <>
   StringReplace[ToString[expr], "$" → ""] <> "\");};";
Returns[expr_ → res_] := "typedef " <> StringReplace[ToString[res],
    {"$" → "", "[" → "<", "]" → ">::res ",
     a : (WordCharacter ~~ (WordCharacter ..)) → "typename " <> a}] <> " res;};" <>
   "\n" <> "\n";
Returns[expr_] := "typedef " <> StringReplace[ToString[expr],
    {"[" → "<", "]" → "> ", "_" → ""}] <> " res;};" <> "\n" <> "\n";
Whole[expr_] := Specialisation[expr] <> "\n" <> SecondRow[expr] <>
   "\n" <> PrintInfo[expr] <> "\n" <>Returns[expr];
WholeList[a_] := StringJoin@@ (Whole /@ a);
```

So, I'm lazy, and I've added some requirements to the Mathematica specification programs (these can be easily omitted, but as I said, I'm lazy). So, the pattern placeholders must be exactly 1 character.

Example: This is ok:

```
m_
```

This is not:

```
expr_
```

Also, the type names must be at least 2 characters, and if you want to return a specific type, you should add $ between its letters. For example, if you want your function to return False, you'll actually write False. It's a bit strange, but I only needed it for the AND function, so never really felt how terrible it was.

Now, let's do some Lambda Calculus implementation. I like the Lambda, because it's so easy to parse and evaluate, but is powerful enough to be a Turing-complete. The specification of the evaluation of Lambda expressions is just a Mathematica list, which we pass to the WholeList to do the translation for us.

Here is the specification:

```
list = {
   Pair[a_, b_],
   Valu[a_],
   Expr[a_],
   Lamb[a_, b_],
   Appl[a_, b_],
   Eval[e_, m_],
   Bred[p_, m_],
   Free[x_, e_],
   IF[c_, t_, e_],
   IF[True, t_, e_] → t,
   IF[False, t_, e_] → e,
   EtaRed[a_, b_, c_, d_],
   AND[a_, b_],
   AND[True, True] → T$r$u$e,
   AND[True, False] → F$a$l$s$e,
   AND[False, True] → F$a$l$s$e,
   AND[False, False] → F$a$l$s$e,
   Eval[Valu[a_], Null] → Valu[a],
   Eval[Valu[a_], Pair[b_, c_]] → Eval[Valu[a], c],
   Eval[Valu[a_], Pair[Pair[Valu[a_], b_], c_]] → b,
   Eval[Lamb[v_, e_], m_] → Lamb[v, Eval[e, m]],
   Eval[Expr[a_], m_] → Eval[a, m],
   (*β-reduction*)
   Bred[Pair[a_, b_], m_] → Appl[a, b],
   Bred[Pair[Lamb[Valu[a_], b_], c_], m_] → Eval[b, Pair[Pair[Valu[a], c], m]],
   Eval[Appl[l_, f_], m_] → Bred[Pair[Eval[l, m], Eval[f, m]], m],
   (*finding if a variable is free*)
   Free[Valu[a_], e_] → True,
   Free[Valu[a_], Valu[a_]] → False,
   Free[Valu[a_], Expr[e_]] → Free[Valu[a], e],
   Free[Valu[a_], Lamb[Valu[a_], e_]] → True,
   Free[Valu[a_], Lamb[v_, e_]] → Free[Valu[a], e],
   Free[Valu[a_], Appl[x_, y_]] → AND[Free[Valu[a], x], Free[Valu[a], y]],
   (*η-reduction*)
   EtaRed[Valu[a_], f_, b_, m_] → IF[Free[Valu[a], f], Eval[f, m], Lamb[Valu[a], Appl[Eval[f, m], b]]],
   Eval[Lamb[Valu[a_], Appl[f_, g_]], m_] → EtaRed[Valu[a], f, Eval[g, m], m]
  };
```

It's not something big, but it's bigger than the translator. Now feed the WholeList and watch what happens: a rabbit comes out:

```
WholeList[list] // StringForm
```

```
template<typename a, typename b>
struct Pair{
static void prt(void){printf("Pair[a_, b_]");};
typedef Pair<a, b>  res;};

template<typename a>
```

I've omitted the contents, because, yes, they're long, and I'm going to copy them form Mathematica right into Visual Studio:

```
#include <iostream>
using namespace std;

struct Null{
	static void prt(void){printf("Null");};
	typedef Null res;
};
struct True{
	static void prt(void){printf("True");};
	typedef True res;
};
struct False{
	static void prt(void){printf("False");};
	typedef False res;
};
struct x{};
struct y{};
struct z{};

//Mathematica generated code
template<typename a, typename b>
struct Pair{
static void prt(void){printf("Pair[a_, b_]");};
typedef Pair<a, b>  res;};

template<typename a>
struct Valu{
static void prt(void){printf("Valu[a_]");};
typedef Valu<a>  res;};

template<typename a>
struct Expr{
static void prt(void){printf("Expr[a_]");};
typedef Expr<a>  res;};

template<typename a, typename b>
struct Lamb{
static void prt(void){printf("Lamb[a_, b_]");};
typedef Lamb<a, b>  res;};

template<typename a, typename b>
struct Appl{
static void prt(void){printf("Appl[a_, b_]");};
typedef Appl<a, b>  res;};
```

```
template<typename e, typename m>
struct Eval{
static void prt(void){printf("Eval[e_, m_]");};
typedef Eval<e, m>  res;};

template<typename m, typename p>
struct Bred{
static void prt(void){printf("Bred[p_, m_]");};
typedef Bred<p, m>  res;};

template<typename e, typename x>
struct Free{
static void prt(void){printf("Free[x_, e_]");};
typedef Free<x, e>  res;};

template<typename c, typename e, typename t>
struct IF{
static void prt(void){printf("IF[c_, t_, e_]");};
typedef IF<c, t, e>  res;};

template<typename e, typename t>
struct IF<True, t, e> {
static void prt(void){printf("IF[True, t_, e_] -> t");};
typedef t res;};

template<typename e, typename t>
struct IF<False, t, e> {
static void prt(void){printf("IF[False, t_, e_] -> e");};
typedef e res;};

template<typename a, typename b, typename c, typename d>
struct EtaRed{
static void prt(void){printf("EtaRed[a_, b_, c_, d_]");};
typedef EtaRed<a, b, c, d>  res;};

template<typename a, typename b>
struct AND{
static void prt(void){printf("AND[a_, b_]");};
typedef AND<a, b>  res;};

template<>
struct AND<True, True> {
static void prt(void){printf("AND[True, True] -> True");};
typedef True res;};

template<>
struct AND<True, False> {
static void prt(void){printf("AND[True, False] -> False");};
typedef False res;};

template<>
struct AND<False, True> {
static void prt(void){printf("AND[False, True] -> False");};
typedef False res;};

template<>
struct AND<False, False> {
```

```
static void prt(void){printf("AND[False, False] -> False");};
typedef False res;};

template<typename a>
struct Eval<Valu<a> , Null> {
static void prt(void){printf("Eval[Valu[a_], Null] -> Valu[a]");};
typedef typename Valu<a>::res  res;};

template<typename a, typename b, typename c>
struct Eval<Valu<a> , Pair<b, c> > {
static void prt(void){printf("Eval[Valu[a_], Pair[b_, c_]] -> Eval[Valu[a],
c]");};
typedef typename Eval<typename Valu<a>::res , c>::res  res;};

template<typename a, typename b, typename c>
struct Eval<Valu<a> , Pair<Pair<Valu<a> , b> , c> > {
static void prt(void){printf("Eval[Valu[a_], Pair[Pair[Valu[a_], b_], c_]] ->
b");};
typedef b res;};

template<typename e, typename m, typename v>
struct Eval<Lamb<v, e> , m> {
static void prt(void){printf("Eval[Lamb[v_, e_], m_] -> Lamb[v, Eval[e,
m]]");};
typedef typename Lamb<v, typename Eval<e, m>::res >::res  res;};

template<typename a, typename m>
struct Eval<Expr<a> , m> {
static void prt(void){printf("Eval[Expr[a_], m_] -> Eval[a, m]");};
typedef typename Eval<a, m>::res  res;};

template<typename a, typename b, typename m>
struct Bred<Pair<a, b> , m> {
static void prt(void){printf("Bred[Pair[a_, b_], m_] -> Appl[a, b]");};
typedef typename Appl<a, b>::res  res;};

template<typename a, typename b, typename c, typename m>
struct Bred<Pair<Lamb<Valu<a> , b> , c> , m> {
static void prt(void){printf("Bred[Pair[Lamb[Valu[a_], b_], c_], m_] ->
Eval[b, Pair[Pair[Valu[a], c], m]]");};
typedef typename Eval<b, typename Pair<typename Pair<typename Valu<a>::res ,
c>::res , m>::res >::res  res;};

template<typename f, typename l, typename m>
struct Eval<Appl<l, f> , m> {
static void prt(void){printf("Eval[Appl[l_, f_], m_] -> Bred[Pair[Eval[l, m],
Eval[f, m]], m]");};
typedef typename Bred<typename Pair<typename Eval<l, m>::res , typename
Eval<f, m>::res >::res , m>::res  res;};

template<typename a, typename e>
struct Free<Valu<a> , e> {
static void prt(void){printf("Free[Valu[a_], e_] -> True");};
typedef typename True res;};

template<typename a>
struct Free<Valu<a> , Valu<a> > {
```

```
static void prt(void){printf("Free[Valu[a_], Valu[a_]] -> False");};
typedef typename False res;};

template<typename a, typename e>
struct Free<Valu<a> , Expr<e> > {
static void prt(void){printf("Free[Valu[a_], Expr[e_]] -> Free[Valu[a],
e]");};
typedef typename Free<typename Valu<a>::res , e>::res  res;};

template<typename a, typename e>
struct Free<Valu<a> , Lamb<Valu<a> , e> > {
static void prt(void){printf("Free[Valu[a_], Lamb[Valu[a_], e_]] -> True");};
typedef typename True res;};

template<typename a, typename e, typename v>
struct Free<Valu<a> , Lamb<v, e> > {
static void prt(void){printf("Free[Valu[a_], Lamb[v_, e_]] -> Free[Valu[a],
e]");};
typedef typename Free<typename Valu<a>::res , e>::res  res;};

template<typename a, typename x, typename y>
struct Free<Valu<a> , Appl<x, y> > {
static void prt(void){printf("Free[Valu[a_], Appl[x_, y_]] ->
AND[Free[Valu[a], x], Free[Valu[a], y]]");};
typedef typename AND<typename Free<typename Valu<a>::res , x>::res , typename
Free<typename Valu<a>::res , y>::res >::res  res;};

template<typename a, typename b, typename f, typename m>
struct EtaRed<Valu<a> , f, b, m> {
static void prt(void){printf("EtaRed[Valu[a_], f_, b_, m_] ->
IF[Free[Valu[a], f], Eval[f, m], Lamb[Valu[a], Appl[Eval[f, m], b]]]");};
typedef typename IF<typename Free<typename Valu<a>::res , f>::res , typename
Eval<f, m>::res , typename Lamb<typename Valu<a>::res , typename
Appl<typename Eval<f, m>::res , b>::res >::res >::res  res;};

template<typename a, typename f, typename g, typename m>
struct Eval<Lamb<Valu<a> , Appl<f, g> > , m> {
static void prt(void){printf("Eval[Lamb[Valu[a_], Appl[f_, g_]], m_] ->
EtaRed[Valu[a], f, Eval[g, m], m]");};
typedef typename EtaRed<typename Valu<a>::res , f, typename Eval<g, m>::res ,
m>::res  res;};
//End of Mathematica generated code

typedef Lamb<Valu<x>,Valu<x> > l1;
typedef Appl<l1, Valu<y> > l2;
typedef Eval<l2, Null>::res l3;
typedef Lamb<Valu<x>, Appl<Valu<x>, Valu<x> > > omega;
typedef Appl<omega, omega> Omega;
typedef Eval<Omega, Null>::res br;
int main(){
      l1::prt();
      printf("\n");
      l2::prt();
      printf("\n");
      l3::prt();
      printf("\n");
      omega::prt();
```

```
        printf("\n");
        Omega::prt();
        printf("\n");
        br::prt();
        printf("\n");
        system("pause");
        return 0;
}
```

But watch out with this code, it's dangerous! I'll advice you to save the cpp file first.

Now compile it!

Hmm, strange, it blows off the compiler! Mine suffers an internal error! What have I done! There must be a mistake somewhere.

I dare you to find it…

Just kidding, everything works just fine, and if the compiler hadn't crushed I would have been pretty sure that there's a mistake somewhere.

Why did it crashed? The Omega is the good-old Omega non-evaluable Lambda expression, yes, it is:

$$\left((\lambda x.\,(x\ x))(\lambda x.\,(x\ x))\right)$$

References:

Koskinen J. – Metaprogramming in C++