Using FPLs to Facilitate C++MP

Seyed Hossein Haeri
Sibylle Schupp
Jonathan Hüser

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH)
Hamburg, Germany

09 Sep 2012
Road Map

- Walk Through Commonalities and Differences between C++ MP and FPLs
- Explore the Facilitations and Impediments for (Semi-) Automatic Translation
- Alternative Solutions
Generic Programming

- Not as natural as OOP – especially $\Rightarrow$ Generative Programming
- Tool support not as good as OOP
- C++MP
  - used for both Generic and Generative Programming
  - known for a long time (since 1998?) to have a lot in common with FP
  - [ Aus98, Ale01, AG04, Gol, SZ09, Sin11a, Sin11b, SS11, Mil ]
How can FPLs be of help?

- Syntax and Error Messages
  - C++: never designed for C++MP ⇒ spontaneous and awkward
  - FPLs: typically ship with a neat syntax
  ⇒ Why not use FPLs to facilitate C++MP?
- Here: Scala and C++MP, paper: (Haskell and F# too)
GCD

C++MP:

```cpp
template<long unsigned a, long unsigned b>
struct GCD{const static long unsigned value = GCD<b, a % b>::value;};

template<long unsigned a>
struct GCD<a, 0>{const static long unsigned value = a;};
```

Scala:

```scala
def gcd: (Char, Char) => Char =
    case (a, b) if (b == 0) => a
    case (a, b) => gcd(b, (a % b).toChar)
```

Observe:

- Immutability
- Order of Pattern Match
  - Wrong? Not necessarily warning/error
  - Loop Forever
- Only unsigned in Scala: Char
GCD

C++MP:

```cpp
template<long unsigned a, long unsigned b>
struct GCD{const static long unsigned value = GCD<b, a % b>::value;};

template<long unsigned a>
struct GCD<a, 0>{const static long unsigned value = a;};
```

Scala:

```scala
def gcd: (Char, Char) => Char =
  case (a, b) if (b == 0) => a
  case (a, b) => gcd(b, (a % b).toChar)
```

Observe:

- Immutability
- Order of Pattern Match
  - Wrong? Not necessarily warning/error
  - Loop Forever
- Only unsigned in Scala: Char
Representation

template<
    long unsigned p,
    long unsigned q = 1,
    bool negative = false>
struct Rational {

    BOOST_STATIC_ASSERT((q != 0));

    const static long unsigned gcd = GCD<p, q>::value;
    typedef mpl::integral_c<long unsigned, p / gcd> num_t;
    typedef mpl::integral_c<long unsigned, q / gcd> den_t;

    static string to_string()
    {
        return p? (negative? "-": "") +
            lexical_cast<string>(num_t::value) +
            (den_t::value == 1?
                ": "/ + lexical_cast<string>(den_t::value))
        ): "0";
    }
};

case class FractionS(
    p: Char,
    q: Char = 1,
    neg: Boolean = false)
{

    require(q != 0)

    private val g = gcd(p, q)
    val num: Char = (p / g).toChar
    val den: Char = (q / g).toChar

    override def toString() =
        (if (neg) "-" else ") +
            num.toByte +
            (if (den == 1) ": "/ + den.toByte)

    }

Observe:

- BOOST_STATIC_ASSERT vs require
- ?: vs if ... else ... 
- Object Encapsulation
Representation

```cpp
template<
    long unsigned p,
    long unsigned q = 1,
    bool negative = false>
struct Rational {

    BOOST_STATIC_ASSERT((q != 0));

    const static long unsigned gcd = GCD<p, q>::value;
    typedef mpl::integral_c<long unsigned, p / gcd> num_t;
    typedef mpl::integral_c<long unsigned, q / gcd> den_t;

    static string to_string() {
        return p? (negative? "-" : "") +
            lexical_cast<string>(num_t::value) +
            (den_t::value == 1?
                "": "/" + lexical_cast<string>(den_t::value))
        : "0";
    }
};
```

```scala
case class FractionS(
    p: Char,
    q: Char = 1,
    neg: Boolean = false){

    require(q != 0)

    private val g = gcd(p, q)
    val num: Char = (p / g).toChar
    val den: Char = (q / g).toChar

    override def toString() =
        (if (neg) "-" else "") +
        num.toByte +
        (if (den == 1) "" else "/" + den.toByte)
}
```

Observe:

- `BOOST_STATIC_ASSERT` vs `require`
- `?:` vs `if ... else ...`
- Object Encapsulation
Plus (and Minus) 1

```cpp
template
<
  long unsigned p1, long unsigned q1,
  long unsigned p2, long unsigned q2, bool n
>
struct Plus<Rational<p1, q1, n>, Rational<p2, q2, n> >
{
  typedef typename Rational<p1, q1, n>::num_t num1_t;
  typedef typename Rational<p2, q2, n>::num_t num2_t;
  typedef typename Rational<p1, q1, n>::den_t den1_t;
  typedef typename Rational<p2, q2, n>::den_t den2_t;

  typedef Rational
  <
    num1_t::value * den2_t::value +
    num2_t::value * den1_t::value,
    den1_t::value * den2_t::value,
    n
  > type;
};
```

```scala
def plus: (FractionS, FractionS) => FractionS = {
  case (r1 @ FractionS(_, _, n1),
    r2 @ FractionS(_, _, n2))
    if n1 == n2 =>
      val num1 = r1.num
      val den1 = r1.den
      val num2 = r2.num
      val den2 = r2.den

      FractionS((num1 * den2 + num2 * den1).toChar,
                 (den1 * den2).toChar,
                 n1)

  ...}
```

Note:

- Named Return Entities
- Underscore Usage in Scala
- Extra Name-Binding to Avoid Repetition ⇒ Brevity!
Plus (and Minus) 1#2

```cpp
#include <iostream>
#include <limits>

template <
    long unsigned p1, long unsigned q1,
    long unsigned p2, long unsigned q2, bool n>
struct Plus<Rational<p1, q1, n>, Rational<p2, q2, n> >
{
    typedef typename Rational<p1, q1, n>::num_t num1_t;
    typedef typename Rational<p2, q2, n>::num_t num2_t;
    typedef typename Rational<p1, q1, n>::den_t den1_t;
    typedef typename Rational<p2, q2, n>::den_t den2_t;

typedef Rational<
    num1_t::value * den2_t::value +
    num2_t::value * den1_t::value,
    den1_t::value * den2_t::value,
    n
> type;
};
```

```scala
def plus: (FractionS, FractionS) => FractionS = {
    case (r1 @ FractionS(_, _, n1),
         r2 @ FractionS(_, _, n2))
        if n1 == n2 =>
            val num1 = r1.num
            val den1 = r1.den
            val num2 = r2.num
            val den2 = r2.den
            FractionS((num1 * den2 + num2 * den1).toChar,
                       (den1 * den2).toChar,
                       n1)
    ...
```

Note:

- Named Return Entities
- Underscore Usage in Scala
- Extra Name-Binding to Avoid Repetition ⇒ Brevity!
template <
  long unsigned p1, long unsigned q1,
  long unsigned p2, long unsigned q2, bool n
>
struct Plus<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{
  typedef typename mpl::if_c
  <
    n,
    typename Minus<Rational<p2, q2, !n>,
      Rational<p1, q1, !n> >::type,
    typename Minus<Rational<p1, q1, n>,
      Rational<p2, q2, n> >::type
  >::type type;
};

... case (FractionS(p1, q1, n1),
         FractionS(p2, q2, n2))
         
       if n1 != n2 =>

        if(n1) minus(FractionS(p2, q2, n2),
                        FractionS(p1, q1, n2))
      else minus(FractionS(p1, q1, n1),
                        FractionS(p2, q2, n1))

   }

Note:
- Mutual Recursion
- Nested Entity Forwarding
template <
  long unsigned p1, long unsigned q1,
  long unsigned p2, long unsigned q2, bool n
>
struct Plus<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{
  typedef typename mpl::if_c
  <
    n,
    typename Minus<Rational<p2, q2, !n>,
      Rational<p1, q1, !n> >::type,
    typename Minus<Rational<p1, q1, n>,
      Rational<p2, q2, n> >::type
  >::type type;
};

...  
case ( 
    FractionS(p1, q1, n1),
    FractionS(p2, q2, n2))
if n1 != n2 =>
  if(n1) minus(FractionS(p2, q2, n2),
               FractionS(p1, q1, n2))
else minus(FractionS(p1, q1, n1),
          FractionS(p2, q2, n1))
};

Note:

- Mutual Recursion
- Nested Entity Forwarding
## Summary

<table>
<thead>
<tr>
<th>C++MP</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. general case first, ordering of the rest irrelevant</td>
<td>S(FPMC) X</td>
</tr>
<tr>
<td>2. static assertion for partially defined ADTs</td>
<td>● X</td>
</tr>
<tr>
<td>3. object encapsulation for compile-time ADTs</td>
<td>● X</td>
</tr>
<tr>
<td>4. default values for template parameters</td>
<td>● X</td>
</tr>
<tr>
<td>5. compile-time data members</td>
<td>● X</td>
</tr>
<tr>
<td>6. named nested entities(●)</td>
<td>S(BSFD) X</td>
</tr>
<tr>
<td>7. type representation for values vs real nested values</td>
<td>S(○) X</td>
</tr>
<tr>
<td>8. mixed preprocessor/template C++MP</td>
<td>S(○) M</td>
</tr>
<tr>
<td>9. no relative ordering between specialisations</td>
<td>S(FPMC) M</td>
</tr>
</tbody>
</table>

○ = No Direct Support, ● = Direct Support Available, ○ = Indirect Support Available, S = Semi-Automatically Possible,
FPMC = first pattern match chosen, BSFD = built-in support for specifying function deliverables, X = Syntactic, M = Semantic
Lessons Taken

(Semi-) Automatic Translation:

- huge human involvement
- error-prone, if scalable at all, for mixed preprocessor/template C++MP
- likely to cater subtle bugs for pattern-matching differences

⇒ Better option (perhaps): a front-end DSL with
  - built-in FP support
  - a terse Scala-like syntax in closer proximity to C++
  - a deliberate mimic of the preprocessor/template mixture
  - same unusual pattern-matching mechanism as C++MP
  - same unusual mixture of laziness and strictness as C++ templates
Lessons Taken

(Semi-) Automatic Translation:

- huge human involvement
- error-prone, if scalable at all, for mixed preprocessor/template C++MP
- likely to cater subtle bugs for pattern-matching differences

⇒ Better option (perhaps): a front-end DSL with
  - built-in FP support
  - a terse Scala-like syntax in closer proximity to C++
  - a deliberate mimic of the preprocessor/template mixture
  - same unusual pattern-matching mechanism as C++MP
  - same unusual mixture of laziness and strictness as C++ templates
Questions?
Mixed Preprocessor/Template Metaprogramming

```cpp
#define OP_INIT_TEMPLATES(r, NestedType, OpName) 
  template<typename T1, typename T2> 
  struct OpName 
  {
    typedef typename OpName
    <
      typename T1::NestedType,
      typename T2::NestedType
    >::NestedType NestedType;
  };

template
<
  typename T1, long unsigned p1,
  long unsigned q1,
  bool n1,
  typename T2
>
struct OpName<Rational<p1, q1, n1>, T2> 
{
  typedef typename OpName
  <
    Rational<p1, q1, n1>,
    typename T2::NestedType
  >::NestedType NestedType;
};

#define RATIONAL_OPS
  (Less, (EqualTo, (Multiplies,
    (Plus, (Minus, BOOST_PP_NIL)))))

BOOST_PP_LIST_FOR_EACH(OP_INIT_TEMPLATES,
  type, RATIONAL_OPS)
```
Want something more real?

```cpp
#define DUMMY_INDEXER(z, n, data) data[n]
#define CALL_WRAPPER_MACRO(z, n, unused) \
    template<> struct S<n>{ \
        template<typename Array> \
        double operator () (const Array& a) \
        {return f(BOOST_PP_ENUM(n, DUMMY_INDEXER, a));} \
    };
```
Quiz

What’s the C++MP equivalent of the following Haskell?

```haskell
r1 = FractionH 2 3 False
r2 = FractionH 1 3 True
r3 = FractionH 5 3 False
r4 = FractionH 10 9 True
r5 = FractionH 3 5 False

result = (plus (multiplies r4 r5) (neg r2)) <
        (minus (divides (neg r4) r1) r3)
```

```cpp
typedef Rational<2, 3> r1t;
typedef Rational<1, 3, true> r2t;
typedef Rational<5, 3> r3t;
typedef Rational<10, 9, true> r4t;
typedef Rational<3, 5> r5t;

typedef LessEqual
        <
        PlusMultiplies<r4t, r5t>, Negate<r2t> >,
        MinusDivides<Negate<r4t>, r1t>, r3t>
>::type result;
```
What’s the C++MP equivalent of the following Haskell?

```cpp
typedef Rational<2, 3> r1t;
typedef Rational<1, 3, true> r2t;
typedef Rational<5, 3> r3t;
typedef Rational<10, 9, true> r4t;
typedef Rational<3, 5> r5t;

typedef LessEqual<
    Plus<Multiplies<r4t, r5t>, Negate<r2t>>,
    Minus<Divides<Negate<r4t>, r1t>, r3t>
>::type result;
```
D. Abrahams and A. Gurtovoy.  
*C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).*  
AW Prof., 2004.

A. Alexandrescu.  
*Modern C++ Design: Generic Prog. and Design Patterns Applied.*  

M.H. Austern.  
*Generic Programming and the STL: Using and Extending the C++ Standard Template Library.*  

S. Golodotz.  
Functional Programming Using C++ Templates (Part 1).  

B. Milewski.
Á. Sinkovics.
Funcional Extensions to the Boost Metaprogram Library.

Á. Sinkovics.
Nested Lamda Expressions with Let Expressions in C++ Template Metaprograms.

S.H. HAERI and S. Schupp.
Functional Metaprogramming in C++ and Cross-Lingual Development with HASKELL.