Using Functional Languages to Facilitate C++ Metaprogramming

Seyed H. HAERI (Hossein), Sibylle Schupp, and Jonathan Hüser

Institute for Software Systems, Hamburg University of Technology, Germany

Abstract. Template and Preprocessor Metaprogramming are both well-known in the C++ community to have much in common with Functional Programming (FP). Recently, very few research threads on underpinning these commonalities have emerged to empower cross-development of C++ Metaprogramming (C++MP) and FP. In this paper, we program a self-contained real-world example in a side-by-side fashion to explore the usefulness of a few mainstream FP languages for this purpose: We develop a compile-time abstract datatype for Rational Numbers in C++. We then present the runtime equivalent in HASKELL, F#, and Scala to discuss some FP parallels across the languages. Here, we consider semi-automatic translation between C++MP and FP languages, for earlier studies on these parallels have already obviated the unsuitability of fully automatic translations. Our study also shows superiority of multiparadigm FP languages over single-paradigm ones. In particular, we conclude Scala to currently be the most promising FP language for facilitating C++MP.

1 Introduction

In 1994, Unruh wrote a C++ program which was designed to emit some prime numbers as warning messages [13]. These prime numbers were calculated at compile-time using techniques which are well-known in today's template metaprogramming. A year later, Veldhuizen introduced expression templates to the world of C++MP [14]. Austern's book [3] exemplified some commonalities between STL (the Generic Programming part of the 1998 C++ Standard [1] library) and FP. Alexandrescu's book presented a *tour de force* of C++MP and was the first to explain some similarities between that and FP. Fast growth of the Boost C++ libraries catered a handful of metaprogramming libraries so well that Abrahams and Gurtovoy devoted their book [2] to that.

Recently, a new trend of FP-*injection* has started to emerge in the C++ community. All that is based on the common belief that C++MP is essentially FP but at the metaprogramming level. Yet, the limited references in support of that belief used to fit into two groups, neither of which suits a new C++ programmer wanting to delve into the topic: Ones which hardly scratch the surface by providing examples on say how to implement simple compile-time functions such as factorial. Or, sizeable manuals of the relevant Boost libraries. Functional programmers approaching C++ to examine that belief face the same difficulty.

Haeri and Schupp [8] were the first to fill this gap. They offer a self-contained real-world explanation of the FP techniques used in C++MP which is short and approachable on the one hand, and inclusive on the other. In this paper, we build on their work

by showing the important bits of a compile-time abstract datatype representing Rational Numbers ($\mathbb{Q}$)[1] in C++. We develop that side-by-side in HASKELL, F#, and Scala. We, furthermore, augment their thread on how FP languages can facilitate C++MP – development of which is well-known to demand a plethora of domain expertise.

We do not aim to provide the most efficient or most facilitating implementation. In fact, both `Rational` of HASKELL and Boost.Ratio outperform our solution from several standpoints. None of the techniques we present are new either. We leverage our compile-time presentation for $\mathbb{Q}$ to demonstrate the FP techniques commonly used in C++MP. So, we go as deep into the technical details as required by a minimalist comparison.

On the other hand, with FP becoming more popular in the C++ community, attempts on automatic translation from HASKELL to C++ are also gaining gravity. For reasons explained by Haeri and Schupp [8], fully automatic solutions are, however, not realistic. Hence, [8] discusses the nuances of a bidirectional translation for semi-automatic cross-lingual development. We build on their detailed comparison by showing how hybrid FP languages, especially Scala, can facilitate C++MP much more naturally.

In our implementation, we call our C++ datatype `Rational`. The HASKELL, F#, and Scala datatypes are named `FractionH`, `FractionF`, and `FractionS`, respectively. This is all to give less grounds for name confusion. In this paper, we do not use the *0x* features of C++. In particular, as opposed to the C++0x *parameter pack* token, ellipsis here is our informal representation of unimportant details. We drop namespace qualifiers (i.e., `std::` and `boost::`) throughout our C++ codes for brevity reasons. Due to space constraints, when the readability is not particularly impinged, we make the code extra compact in common ways. We assume enough familiarity with all the above languages.

On the matter of coding, this paper starts by discussing how to implement the auxiliary functions needed. It then explains how to represent $\mathbb{Q}$ itself in the four languages of discourse. Finally, it explores the implementation techniques we use for operations on $\mathbb{Q}$ in either language. Meanwhile, we run a comparative study of the implementation techniques used across the four languages. With this study of commonalities and differences, we discuss the impediments to automatic cross-lingual development as well as the relative ease of semi-automating this procedure.

This paper starts by reviewing related work in Section 2. Our technical work starts in Section 3 where we compare the FP nature of C++MP with the three FP languages. We then discuss how to represent $\mathbb{Q}$ in Section 4. Section 5.1 discusses how to reduce code repetition solely in C++MP. Next, Sections 5.2 and 5.3 discuss how to implement comparatives and arithmetics. Detailed discussion, finally, follows in the conclusion.

2 Related Work

Sipos et al. [12] start by enumerating some similarities between template metaprogramming and FP. They then informally describe a method for systematically producing metafunctions out of functions written in the pure FP language CLEAN [6]. They advertise that their `Eval<>` metafunction evaluates the produced metaprograms according to the operational semantics of CLEAN. As discussed in [8], whilst they do not formally

[1] For simplicity, in this paper, we do not distinguish between the Set-Theoretical notion of Rational Numbers and our codes representing that for programming purposes.

present their operational semantics, their informal explanation suggests remarkable differences between the operational semantics of CLEAN and that of theirs.

Haeri and Schupp [8] are the first to provide a real-world stand-alone exemplification of C++MP being Functional in nature. They explore impediments against fully-automatic cross-lingual development between C++MP and HASKELL. Armed with that, they suggest the approachability of a semi-automatic cross-lingual development between C++MP and hybrid FP languages.

Sinkovics [9] offers certain solutions for improving the FP support in Boost.MPL and discusses why they are needed. Sinkovics and Porkoláb [11] advertise implementation of a λ-library on top of the operational semantics of Sipos et al. for embedded FP in C++. They also later advertise [15] extension of their λ-library to full support for HASKELL. None of their works are unfortunately available online for experimentation. We can therefore not evaluate their works any further. [2] Sinkovics [10] offers a restricted solution for emulating let-bindings in template metaprogramming.

Milewski [4] has a number of posts on his personal blog that speak about Monads [5], their benefits for C++, and how to implement Monadic entities in C++. He also has a post on how template metaprogramming with the newly added C++ construct *variadic templates* is similar to lazy list processing in HASKELL. Finally, Sankel [7] shows how to implement algebraic datatypes in C++.

3 Greatest Common Divisor

Fraction cancellation is the cornerstone of any arithmetic on $\mathbb{Q}$ – that is, dividing the nominator and denominator by their greatest common divisor (*gcd*). The metafunction implementation of *gcd* using Euclid's famous algorithm is a straightforward pattern matching – the first FP technique used in C++MP that we present:

```
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;};
```

This is one of the very few examples where the mathematical definition of an operation is not far from its C++ metafunction implementation. The HASKELL implementation is even better and is literally *identical* to Euclid's algorithm. Both implementations use tail recursion, obviously on immutable data:

```
gcd a 0 = a; gcd a b = gcd b (a `mod` b)
```

The F# version also uses pattern matching, and is quite more verbose in this case – although not as verbose as the C++ one. We postpone the Scala version until Section 4 where we discuss how to represent the abstract datatype.

```
let rec gcd (a: uint32) (b: uint32) = match b with | 0u -> a | b -> gcd b (a % b)
```

[2] Through personal email exchange, we were informed that they are elaborating on their developments. The result is to be placed online for experimentation.

An important difference between C++ and FP languages here is that, in C++, the general definition of a template needs to be introduced first. Only then can the specialisations come but their relevant order of definition is not significant so long as they are in scope. In FP languages, however, the order of pattern matches is significant and that does include the general case. Here, for instance, the general case needs to come after the case for $gcd(a,0)$, or the recursion will never end.

We use the following example to demonstrate the significance of this difference in action when considering automatic translation. Let us suppose – for the sake of argument – that $gcd(0,a) = 0$. In order to accommodate that, one would need to also provide the following two specialisations for the C++ metafunction:

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

and the order of the template specialisations is not relevant because all the in-scope specialisations will be equally visible at the instantiation time. On the other hand, for the HASKELL counterpart, the following two cases would have needed to be added **before** the general case, for HASKELL always goes for the first pattern match:

```
gcd 0 0 = 0; gcd 0 a = 0
```

As discussed in Section 5.2, this subtle difference can under certain circumstances cause compilation failures upon automatic translation from HASKELL to C++. However, as far as the relative order of definitions is the concern, the real notoriety faced for such a translation is right here. Consider a translation from C++ to HASKELL which naïvely adds the two new *gcd* cases **after** the existing cases, and in particular, after the general case: No error/warning messages may be emitted in either language; both implementations are furthermore logically correct; yet, the subtle difference between languages entails observing different results. The reason will be arduous to spot and is likely to cater a full genre of implementation bugs. Automatic translation from C++ to HASKELL will be even trickier for the *correct* order of patterns needs to be figured out.

4 Representing Rational Numbers

For a compile-time representation of $\mathbb{Q}$, the nominator and denominator need to be stored as template parameters. Note that C++ template metaprograms are **pure** functional entities. As a result, the passed template arguments cannot be mutated, and the cancelled nominator and denominator have to be stored as nested values/types. We choose to represent those using types to avoid possible linkage failures:

```
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";
  }
};
```

In order to give equal range to the nominator and denominator, we use unsigned types for both. That entails storing the sign of a fraction in a separate Boolean template parameter (`negative`). [3] The Scala version closely takes after its C++ counterpart:

```
case class FractionS(n: Char, d: Char = 1, neg: Boolean = false){
  require(d != 0)
  private val g = gcd(n, d)
  val num: Char = (n / g).toChar
  val den: Char = (d / g).toChar

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

  private def gcd (a: Char, b: Char): Char =
    if (b == 0) a else gcd(b, (a % b).toChar)
}
```

It is noteworthy that the only unsigned integral type in Scala is `Char` which is only two bytes wide – not scaling at all to its C++ counterpart. Developing new types with near native support is relatively easy in Scala. Yet, we employ this example here to remind that whether types get mapped correctly over translation is an important question in the checklist. Not enough scrutiny can give birth to very subtle bugs here.

The fact that `gcd` is a private member function here is remarkable. With OOP being one of the major C++ paradigms, this is most likely to be practised in C++MP likewise. Yet, we chose not to discuss that in Section 3 to stay focused.

The F# version is not as close to C++ as the Scala. Local `let`-bindings are only internally available. Hence, getters `num`, `den`, and `negative` are needed for outside queries. Furthermore, because functions cannot have default values in F#, one needs to add two constructor overloads which relegate the task to the main one (lines 6 and 7 below). Finally, static safety nets such as the Boost assertion or Scala's `require` are not available in F#, and one resorts to (runtime) exceptions thrown in constructors (line 4).

[3] Had we chosen the nominator to carry the sign of the fraction by making it signed, the nominator's maximum absolute value was enforced to be half that of the denominator.

```
type FractionF (p: uint32, q: uint32, n: bool) =
  let gcd = gcd p q
  let num_ = p / gcd
  let den_ = if q = 0u then failwith "division by zero" else q / gcd

  new (p: uint32, q: uint32) = FractionF(p, q, false)
  new (p: uint32) = FractionF(p, 1u, false)

  member this.num with get() = num_
  member this.den with get() = den_
  member this.negative with get() = n

  override this.ToString() = ...
```

The state of affairs is categorically different in HASKELL. This is because, in HASKELL, data structures are represented using pure algebraic datatypes:

```
data FractionH = FractionH Word Word Bool; FractionH _ 0 _ = undefined
```

The first important difference between the C++ version and the HASKELL one is right in how they deal with partially defined datatypes. In C++, one usually employs static assertion to outlaw undefined instantiations. (See the use of `BOOST_STATIC_ASSERT` at line 3 in the definition of `Rational`.) Such a mechanism is not present in HASKELL, so, the second part of `Fraction`'s definition will not prevent its use with 0 as the denominator. HASKELL however does support partial definition of functions. Thus, one way to circumvent this problem is to encapsulate the outlawing in a pivotal function:

```
cancel (FractionH p 0 _) = undefined
cancel (FractionH p q n) = FractionH (p `div` g) (q `div` g) n where g = my_gcd p q
```

Note that, unlike C++, Scala, and F#, there is no way to provide default values for the denominator and sign parameters in HASKELL. Furthermore, because HASKELL does not support OOP, there is no way to store the cancelled nominators and denominators in the body of the type `Fraction`. Likewise, in HASKELL, one may go about string representation of a fraction as follows – again, not encapsulated in `Fraction`:

```
instance Show FractionH where show = show' . cancel where
  show' (FractionH 0 _ _) = "0"
  show' (FractionH p q n) =
    (if n then "-" else "") ++ show p ++ (if q == 1 then "" else "/" ++ show q)
```

The routine string representation is similar across the languages: A 0-nominator fraction is always written as "0" with no sign; when the denominator of a fraction is "1", only the nominator is written; the sign is only written if the fraction is negative.

Due to the lack of local storage in HASKELL, the pattern of cancelling fractions before arithmetic operations will occur repeatedly. Obviously, this on-the-fly cancellation is unacceptable in C++ where efficiency is critical. Cancellation is an example of when, in the translation from HASKELL to C++, one would need to encapsulate the functionality in a (data) member for efficiency reasons. We anticipate that figuring out when similar encapsulations are needed is a highly non-trivial task for automatic translation, let alone the correct encapsulation.

In C++, the detachment of the string representation from the `Rational` class is considered inappropriate because that would be unreasonably scattered. Therefore, an automatic translation from HASKELL to C++ would need to be able to infer when to encapsulate such a function in the class. Likewise, because this encapsulation is not directly possible in HASKELL, one needs to detach similar member functions upon an automatic translation from C++ to HASKELL.

We end this subsection by two basic HASKELL functions which will be handy later:

```
num (FractionH p _ _) = p; den (FractionH _ q _) = q
```

5 Operations

C++ was never specifically designed to be a language for metaprogramming. A consequence is that metafunctions do not come with a built-in *return* mechanism. Hence, one needs to specify the deliverables using **named** nested types/values that are to be queried later. We did already see the use of nested values for returning the value of a *gcd*. Here, we are going to use a type to represent the result of an operation on $\mathbb{Q}$:

```
template<...> struct Plus{typedef Rational<...> type;};
```

An automatic translation from FP languages to C++ needs to be particularly careful on whether to map a given value to a nested value or a nested type.

5.1 Reducing Repetition Using Preprocessor Metaprogramming

Having to specify deliverables using nested entities makes dealing with expression combination very labouring. For example, if we choose to implement plus like

```
template<long unsigned p1, long unsigned q1, bool n1,
         long unsigned p2, long unsigned q2, bool n2> struct Plus {...};
```

there will be no way to perform operations like `Plus<Plus<...>, ...>`. The reason is that `Plus<...>` is not a `Rational` itself – its `type` nested type is. To circumvent this problem, one would provide a general case where both template parameters are arbitrary types (with particular nested types). One would then amend that by the appropriate number of template specialisations.

In our case, for example, one would provide a general `template<typename T1, typename T2> struct Plus` in addition to specialisations for `Plus<T1, Rational<p2, q2, n2> >`, `Plus<Rational<p1, q1, n1>, T2>`, and `Plus<Rational<...>,Rational<...> >`. The trick is that, for all cases, when the template parameter is not a `Rational`, one **forward**s the operation to the respective `type` nested type. The real calculation happens only in the `Plus<Rational<...>, Rational<...> >` case. This repetitive *nested-entity forwarding* will occur for all the $\mathbb{Q}$ operations and C++ programmers often evade similar manual implementations using preprocessor metaprogramming:

```
#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 p2, long unsigned q2, bool n2>\
struct OpName<T1, Rational<p2, q2, n2> >{         \
  typedef typename OpName<typename T1::NestedType,\
                          Rational<p2, q2, n2> >::NestedType NestedType;\
};                                                \
template<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)
```

Here is a quick recap on how that is particularly related to FP: Note first that the list of operation names (`RATIONAL_OPS`) is constructed in exactly the same way as one employs *cons* in FP to construct lists. Next, one needs to note that `BOOST_PP_LIST_FOR_EACH` is in fact the preprocessor equivalent of the famous *map* function in FP.

Given that FP functions do already have built-in support for specifying deliverables, this entire code bloat is redundant when it comes to translation from C++MP to an FP language. Figuring that out does not seem to be an easy task for automatic translation. The other translation direction is in fact not needed to generate the preprocessor portion because a machine can generate all classes themselves without getting bored. Yet, the translator needs to know that this code bloat is indeed needed for metafunctions to enforce nested-entity forwarding.

In sizeable industrial projects where C++MP is needed, a careful mixture of template and preprocessor metaprogramming is often unavoidable. In such cases, deciding on how to deal with each part of the mixture is yet another non-trivial task for automatic translation. See Section 6 for more.

5.2 Comparatives

The general idea to examine equality of two fractions is to examine the respective cancelled nominators and denominators whilst also taking the sign into account. Special cases can be handled quicker. Two fractions can obviously not be equal when they have different signs. However, our cancellation algorithm does not change the original sign of a `Fraction`. There comes a subtle consequence: Although the following two equations hold for every non-zero q:

```
Rational<0, q, false>::num_t::value == Rational<0, q, true>::num_t::value
Rational<0, q, false>::den_t::value == Rational<0, q, true>::den_t::value
```

the fractions would still have different signs. A pointwise equality test will therefore not quite work. The case for two 0 fractions with different signs needs special care. Let us examine the HASKELL implementation first:

```
instance Eq FractionH where
  (FractionH 0 _ _) == (FractionH 0 _ _) = True
  (FractionH _ _ n1) == (FractionH _ _ n2) | (n1 /= n2) = False
  f1 == f2 = (num f1' == num f2' && den f1' == den f2')
    where f1' = cancel f1; f2' = cancel f2
```

Because HASKELL always chooses the first successful pattern match, no conflict happens between the first two cases. The story is different in C++ though for, in C++, there is no relative ordering between the in-scope specialisations. Hence, the following attempt will fail because neither specialisation is a better fit for equality between `Rational<0, q, true>` and `Rational<0, q, false>`. An ambiguity compile error will be emitted for such a comparison attempt. It is noteworthy that an automatic translation from the HASKELL version to C++ is also most likely to produce something like:

```
template<long unsigned q1, bool n1, long unsigned q2, bool n2>
struct EqualTo<Rational<0, q1, n1>, Rational<0, q2, n2> >
{typedef mpl::true_ type;};

template<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct EqualTo<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{typedef mpl::false_ type;};
```

The solution is to merge the two specialisations into one. Expecting an automatic translation from HASKELL to C++ to manage this merging technique is not realistic. It would be even less expectable for the other direction of automatic translation to sort the correct HASKELL ordering out. The case when the two fractions have the same sign is routine and we will present all that together:

```
template<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct EqualTo<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{typedef typename mpl::bool_<p1 == 0 && p2 == 0> type;};
```

```
template<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct EqualTo<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 typename mpl::bool_<
   num1_t::value == num2_t::value && den1_t::value == den2_t::value
  > type;
};
```

In F#, classes with multiple constructors cannot be used for pattern matching. F# programmers circumvent this problem in different ways. Here, we present one of them which turns out to be even terser than the HASKELL version:

```
let equal_to (f1: FractionF) (f2: FractionF) =
  match (f1.num, f1.den, f1.negative), (f2.num, f2.den, f2.negative) with
  | (p1, _, n1), (p2, _, n2) when n1 <> n2 -> p1 = 0u && p2 = 0u
  | (p1, q1, n1), (p2, q2, n2) when n1 = n2 -> p1 = p2 && q1 = q2
```

The simple use of Scala keyword `case` in front of `FractionS` makes it a *case class*. The benefit of that which is particularly significant here is legislating patterns matching based on its values. Despite that, there is a subtlety here that makes the Scala code closer to the C++ one: For each branch, one only queries the cancelled values respective to the branch. (Note that, e.g., no such value was needed for the first branch.)

```
def equal_to: (FractionS, FractionS) => Boolean = {
  case (FractionS(p1, _, n1), FractionS(p2, _, n2)) if n1 != n2 => p1 == 0 && p2 == 0
  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;
    num1 == num2 && den1 == den2
}
```

The main difference between Scala and C++ here is that, like any other FP language, the first pattern match is chosen in Scala. There is one minor difference too: Note the repetition of `Rational<p1, q1, n>` and `Rational<p2, q2, n>` for querying `den1/num1_type` and `den2/num2_type` in C++. This is avoided in Scala by binding `Rational(p1, q1, n1)` and `Rational(p2, q2, n2)` to variables `r1` and `r2`.

In HASKELL where Type Classes are available, our definition of `==` also implements `/=` automatically. Was our `Rational` class a runtime one, C++ Concepts could have likewise been used. Given that `Rational` is for compile-time use, we would need to define a new Concept to be the compile-time counterpart of EqualityComparable. We would like to remind that EqualityComparable defines the types and (runtime) functions each of its instantiations need to provide. In other words, EqualityComparable does not constrain the respective metafunctions of its instantiations. Were we about to go for the metaprogramming counterpart of the `Eq` Type Class of HASKELL, we needed to first define the Concept for types with metafunctions providing a `type` nested-type. Next, we had to define a Concept like MetaEqualityComparable, which states the names of

the relevant equality metafunctions. We would need to implement `NotEqualTo` in terms of `EqualTo` in MetaEqualityComparable and state that `Rational` models it too. The definition of `EqualTo` for `Rational` would then suffice to get `NotEqualTo` automatically. It is also noteworthy that the same discussion applied when one uses *traits* as Type Classes in Scala. We drop the Scala version for its similarity to the HASKELL one.

Alternatively, we can manually define operations in terms of each other without resorting to Concepts. Here, we only present how to do that for `NotEqualTo` in terms of `EqualTo`. Note that in the implementation of `Less`, similar merge techniques to the ones used for `EqualTo` are needed to avoid ambiguity between template specialisations.

```
template<typename T1, typename T2> struct NotEqualTo
{typedef typename mpl::not_<typename EqualTo<T1, T2>::type>::type type;};
```

5.3 Arithmetics

Arithmetic operations on types are typically expected to at least consist of the four basic operations. The metaprogramming techniques used for the implementation are mainly similar. So, we only present plus here in which the main FP technique used is mutual recursion with minus:

```
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;
};
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;
};
```

With the help of elementary calculus, the reader is invited to make their own sense out of the algorithm used above. We drop that routine explanation to save more space.

Perhaps the only syntactic difference between the C++ metaprogram and the HASKELL program is the use of guards in the HASKELL one in contrast to the use of inline opera-

tions. This is because, in HASKELL in order to test whether the two fractions have the same sign or not, we cannot repeat the sign token in the pattern – whereas this is fine in C++. So long as the guard conditions are simple, an automatic translation should not face much difficulty. Yet, complicated guards might need a separate treatment. Despite all that, the two codes are so similar that one could simply rewrite one with the other syntax to got to the other implementation.

```haskell
plus :: FractionH -> FractionH -> FractionH
plus (FractionH p1 q1 n1) (FractionH p2 q2 n2) | n1 == n2 =
  cancel (FractionH (num1 * den2 + num2 * den1) (den1 * den2) n1) where
    f1 = cancel (FractionH p1 q1 n1); f2 = cancel (FractionH p2 q2 n2)
    num1 = num f1; num2 = num f2; den1 = den f1; den2 = den f2
plus (FractionH p1 q1 n1) (FractionH p2 q2 n2) | n1 /= n2 =
  cancel raw_result where raw_result = if n1
    then minus (FractionH p2 q2 n2) (FractionH p1 q1 n2)
    else minus (FractionH p1 q1 n1) (FractionH p2 q2 n1)
```

The subtle difference is that, for better efficiency in the HASKELL version, one would apply a final cancellation to `raw_result` when $n_1 \neq n_2$. However, this is not needed in the C++ version for the cancelled nominators and denominators will be automatically stored upon instantiation. In this very case, automatic translation might not be expected to be able to handle this subtle difference. In larger applications, however, real performance hits upon translation can source from similar subtle differences.

Because Scala automatically stores the cancelled values in local members, final cancelltation is not needed. Roughly speaking, the Scala version is *the compact* C++ except that, like most FP languages, Scala uses guard expressions when repetition is needed in pattern matching. Furthermore, Scala also chooses the first pattern match.

```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)
  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))
}
```

The F# version is roughly of the same length as that of Scala. Yet, it is not as close to the C++ version as the Scala one. Pattern matching on classes is only emulated in F# using queried fields. Hence, pattern matching based on `num`, `den`, and `negative` in one branch makes these readily available to other branches too. It is sensible, thus, to use the same constructs in both branches. This contrasts the C++ and Scala version where branches only query the constructs their respective part of algorithm entails.

Note the use of `let rec ... and ...` in F# for mutual recursion. In C++, forward declaration enables this. No special treatment is needed in HASKELL or Scala.

```
let rec plus (f1: FractionF) (f2: FractionF) =
  match (f1.num, f1.den, f1.negative), (f2.num, f2.den, f2.negative) with
  | (p1, q1, n1), (p2, q2, n2) when n1 = n2 ->
    FractionF(p1 * q2 + p2 * q1, q1 * q2, n1)
  | (p1, q1, n1), (p2, q2, n2) when n1 <> n2 ->
    if n1
    then minus (FractionF(p2, q2, n2)) (FractionF(p1, q1, n2))
    else minus (FractionF(p1, q1, n1)) (FractionF(p2, q2, n1))
and minus f1 f2 = ...
```

5.4 Samples

Complicated arithmetic expressions can now be evaluated using our libraries.

```
typedef Rational<2, 3> r1t; typedef Rational<5, 3> r3t; typedef Rational<3, 5> r5t;
typedef Rational<10, 9, true> r4t; typedef Rational<1, 3, true> r2t;
typedef LessEqual<
  Plus<Multiplies<r4t, r5t>, Negate<r2t> >, Minus<Divides<Negate<r4t>, r1t>, r3t>
>::type result;
```

Note that all computations are compile-time. The corresponding runtime HASKELL is:

```
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)
```

6 Conclusion

In this paper, we show the FP nature of C++MP by demonstrating its proximity to HASKELL, F#, and Scala. We implement a C++ compile-time abstract datatype for $\mathbb{Q}$ which closely corresponds to its FP runtime counterparts, especially Scala. We show that, although the techniques are tightly close, syntax of the FP languages outperforms by far. The proximity has mislead earlier research towards suggesting automatic translation between FP languages and C++MP. Although Haeri and Schupp [8] demonstrate that to be unrealistic, this work shows practicality of semi-automatic translation between C++MP and Scala. Earlier research suggests starting from the FP programs as a design phase and moving to C++MP as the implementation. The idea is that working with the neat syntax of FP is much easier than C++MP, especially for C++ templates are known to be notoriously unapproachable when it comes to error/warning messages. Yet, it is worth noting that software development typically entails several iterations between design and implementation before each release. The ability of going back and forth between the design (FP programs in this case) and the implementation (C++ metaprograms in this case) here becomes vital. We argue therefore that any mechanical translation across the two languages needs to be bidirectional. Fortunately, as shown in this paper, Scala seems promising for this purpose.

With this mindset, alongside the codes that we present in this paper, we consider translations both from FP languages to C++MP and vice versa. We study commonalities which pave the way for mechanical translations, as well as differences as the hindrances on the way. We show how pattern matching, tail recursion, and immutability is

	C++MP	HASKELL	F#	Scala	Section
1	general case first, ordering of the rest irrelevant	S(FPMC)	S(FPMC)	S(FPMC)	3
2	static assertion for partially defined ADTs	C(○)	◐	●	4
3	object encapsulation for compile-time ADTs	S(○)	●	●	4
4	default values for template parameters	N/A(○)	◐	●	4
5	compile-time data members	C(○)	●	●	4
6	named nested entities(◐)	S(BSFD)	S(BSFD)	S(BSFD)	5
7	type representation for values vs real nested values	S(○)	S(○)	S(○)	5
8	mixed preprocessor/template C++MP	S(○)	S(○)	S(○)	5
9	no relative ordering between specialisations	S(FPMC)	S(FPMC)	S(FPMS)	5

○ = No Direct Support, ● = Direct Support Available, ◐ = Indirect Support Available, S = Semi-Automatically Possible, C = Circumventable, FPMC = first pattern match chosen, BSFD = built-in support for specifying function deliverables

Fig. 1. Mechanical Translation between FP Languages and C++MP

closely similar across FP languages and C++MP. Figure 1 summarises the impediments caused by the differences enumerated in this paper. Along with the section visited in, it comments on the feasibility of overcoming each impediment. Here is a row-by-row discussion:

1. The general case should be moved to the top from FP to C++, and to the bottom in the opposite direction of translation. The correct ordering for HASKELL should be provided by the user for the first time and should be retained for further back and forth translation until the next update.

2. From C++ to HASKELL, one needs to resort to partially defined functions with pivotal role, if any. The translation needs to be instructed about the static assertion being artificially encapsulated in a function in the HASKELL equivalent. Use the F# exception mechanism in constructors to emulate the error checking in runtime. Built-in Scala support for static assertion corresponds perfectly. Unless for the exceptional situations for F# and Scala, thus, the translation can be fully automatic.

3. From HASKELL to C++, the translation should be instructed about the functions to be packed together in an abstract datatype. In the other direction, each member function will be translated to a stand-alone function. Both F# and Scala have built-in support here that fit perfectly.

4. Even emulating this is currently not possible in HASKELL. One needs to write as many auxiliary constructors in F# as it takes to redirect the construction to the main one (with the default values set through the redirection). Scala's constructors can have default values and no further intricacies needed. Whilst making the F# translation might sometimes become slightly involved, one can fully automate the Scala translation.

5. From C++ to HASKELL, functions which perform the stored calculation should be contrived to be used on-the-fly every time the compile-time data member is used in the C++ version. The translator needs to be instructed about this correspondence between the data members and functions for next translation iterations. Given the good support of OOP in F# and Scala, the translation can happen automatically for these languages.

6. Use named nested entities in C++, and leave them out in translation to FP languages.

7. From FP languages to C++, the user needs to manually advise the translator about each entity being translated into a nested value or a nested type. For the other direction, every nested entity translates to a value.

8. If possible, from C++ to an FP language, the user is to manually instruct the parts of this mix to dismiss upon translation. From an FP language to C++, the translator needs not to update the dismissed parts. N.B. In many cases, this dismissal might not be possible or very tricky to specify. See the supplementary notes below.

9. When upon the translation from an FP language to C++ a compile error is emitted due this difference, use the merge technique presented in this paper or similar ones in C++. Usages of these techniques need to be marked for the translator to know not to touch the manually corrected translations until there is a change in the FP language. Refer to guideline 1 for retaining the order of specialisations.

This paper makes it obvious that full mechanical translation between Functional programs and C++ metaprograms is not realistic. Nevertheless, Figure 1 and the above discussion demonstrate that semi-automatic translation between C++MP and hybrid FP languages is indeed promising, especially for Scala. On the other hand, from a theoretical viewpoint, C++MP is a lazy FP language with selective strictness. However, as also explained in [8], C++MP is predominantly evaluated strictly as if it only had selective laziness. F# and Scala are both strict FP languages with selective laziness, whilst HASKELL is designed the opposite way around. This is yet another reason to prefer the former two languages over HASKELL (or like-minded languages) for this purpose.

One might here be intrigued to try simulating features of one language in the other one. We would like to remind, however, that the whole point of resorting to FP languages is to harness the spontaneity of C++MP's syntax for FP purposes. In other words, the syntax of C++MP is already exotic enough to make C++ programmers practice a variety of non-trivial indirections. Adding extra layers of indirection on top of that for emulating an FP language would defeat the purpose by entailing extra syntactic chaos. Likewise, because HASKELL is **designed to be uniparadigm**, emulating say OOP often produces very unnatural HASKELL codes. The story is similar for F# and Scala; for instance, levering tricks to manipulate the order in which patterns are matched is not likely to result in natural-feeling code.

Whilst, using an appropriate FP language, one can make semi-automatic translation approachable, some impediments in Figure 1 would still need intensive care. For example, fully removing impediment 8 on mixed template/preprocessor metaprogramming can still be considerably demanding. Below is a simplified instance of where in the industry one of the authors needed to use such a mixture. Consider a C++ struct `S`

templatised by an integer and with a function call operator which, for every n, invokes a fixed function `f` on the first n elements of an array `a` it is called with. It is currently not clear what F# or Scala code can correspond this piece of C++ metaprogram:

```
#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));}               \
  };
```

Our final remark is that the best option is perhaps to design a new FP language with built-in correspondent FP features as that of C++MP itself. A terse syntax based on Scala but with more resemblance to C++ can form a proper surface. A deliberate preprocessor support is also needed for real-world C++ is often full of complex template/preprocessor mixtures. Additionally, the semantics needs to take it into consideration that: C++MP combines laziness and strictness in unusual ways that do not fully correspond to any available FP language; and, pattern matching in C++ uses the best match strategy as opposed to the common first match of FP. A successful such language can then eventually become a part of the C++ tool chain.

References

1. *International Standard ISO/IEC 14882:1998(E): Programming Languages — C++* , (1998).
2. D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, AW Prof., 2004.
3. M.H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, AW Prof. Comp. Series, AW Longman Publ. Co., 1998.
4. B. Milewski, *Bartosz Milewski's Prog. Cafe: Concurrency, Multicore, C++, Haskell*.
5. E. Moggi, *Notions of Computation and Monads*, Inf. & Comp. **93** (1991), no. 1, 55–92.
6. R. Plasmeijer and M. van Eekelen, *Concurrent* CLEAN *Language Report (v 2.0)*, Dec 2001.
7. D. Sankel, *Algebraic Data Types Series*, C++Next: The next generation of C++.
8. S.H. Haeri and S. Schupp, *Functional Metaprogramming in C++ and Cross-Lingual Development with* HASKELL, Tech. report, Uni. Kansas, Oct 2011, Draft Proc. 23^{rd} Symp. Impl. and Appl. Func. Langs., ITTC-FY2012-TR-29952012-01.
9. Á. Sinkovics, *Funcional Extensions to the Boost Metaprogram Library*, Proc. 2^{nd} Works. Generative Tech., Elec. Notes in Theo. Comp. Sci., vol. 264, Jul 2011, pp. 85–101.
10. _____ , *Nested Lamda Expressions with Let Expressions in C++ Template Metaprorgams*, WGT'11 (Z. Porkoláb and N. Pataki, eds.), vol. III, Zolix, 2011, pp. 63–76.
11. Á. Sinkovics and Z. Porkoláb, *Expressing C++ Template Metaprograms as Lambda Expressions*, Trends in Func. Prog. (Z. Horváth, V. Zsók, P. Achten, and P. Koopman, eds.), Trends in Func. Prog., vol. 10, Intellect, UK/The Uni. Chicago Press, USA, June 2–4 2009, pp. 1–15.
12. Á. Sipos, Z. Porkoláb, N. Pataki, and V. Zsók, *Meta<Fun>: Towards a Functional-Style Interface for C++ Template Metaprograms*, Tech. report, Eötvös Loránd Uni, Fac. of Inf., Dept. Prog. Langs., Pázmány Péter sétány 1/C H-1117 Budapest, Hungary, 2007.
13. E. Unruh, *Prime number computation*, 1994, ANSI X3J16-94-0075/ISO WG21-462.
14. T. Veldhuizen, *Expression Templates*, C++ Report **7** (1995), no. 5, 26–31.
15. Z. Porkoláb and Á. Sinkovics, *C++ Template Metaprogramming with Embedded Haskell*, Proc. the 8th Int. Conf. Generative Prog. & Component Engineering (GPCE 2009) (New York, NY, USA), ACM, 2009, pp. 99–108.