Using Functional Languages to Facilitate C++ Metaprogramming

Seyed H. HAERI (Hossein) Sibylle Schupp Jonathan Hüser
Institute for Software Systems, Hamburg University of Technology, Germany
{hossein,schupp,jonathan.hueser}@tu-harburg.de

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 run-time 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 impracticability of fully automatic translations. Our study also shows the 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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

Keywords C++ Metaprogramming, Functional Programming, Scala, F#, HASKELL, Cross-Lingual Development

1. Introduction

In 1994, Unruh wrote a C++ program which was designed to emit some prime numbers as error messages [32]. 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 [34]. Austern's book [4] exemplified some commonalities between STL (the Generic Programming part of the C++ Standard [1, 2] 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 [3] 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 such as MPL[11], Fusion[12], Proto[21], and Preprocessor[15]. Functional programmers approaching C++ to examine that belief face the same difficulty.

Our earlier work [27] fills this gap. We offered a self-contained real-world explanation of the FP techniques used in C++MP, which was short and approachable on the one hand, and inclusive on the other. In this paper too, we show the important bits of a compile-time abstract datatype representing Rational Numbers ($\mathbb{Q}$)[1] in C++. We build on [27] by developing that side-by-side in HASKELL, F#, and Scala. We, furthermore, augment our thread on how FP languages can facilitate C++MP – which is well-known to demand a plethora of domain expertise.

1.1 Motivation

Support for $\mathbb{Q}$ is so important that `Rational` is already a built-in type in HASKELL. For F# too a similar support ships automatically as a part of the Microsoft Solver Foundation [18]. Furthermore, in C++, Boost has libraries for both runtime and compile-time Rational arithmetic. Boost.Rational[22] (for runtime $\mathbb{Q}$ support) is a relatively old hand member of the Boost library. Boost.Ratio[35] has also recently been added to the Boost library to support Rational arithmetics at compile-time. That is all to demonstrate that our use-case here is real-world enough.

However, 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 we explained in [27], fully automatic solutions are, however, not realistic. Hence, [27] discusses the nuances of a bidirectional translation for semi-automatic cross-lingual development. We build on our previous detailed comparison by showing how hybrid FP languages, especially Scala, can facilitate C++MP much more naturally.

This paper serves as a compact real-world tutorial for two groups: the FP programmers wanting to learn C++MP; as well as the C++MP developers who want to learn FP. Not all the prac-

[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.

WGP'12, September 9, 2012, Copenhagen, Denmark.

tical bonus of this paper is this, however. The C++ community is currently looking for ways to reduce the extraordinary development cost of C++MP. To that end, the neat FP syntax/semantics has shown great promise so far. This is the main reason why this paper studies cross-lingual development cycles between C++MP and FP.

Once such cross-lingual development cycles gain enough industrial gravity, a transfer of research ideas from the FP world into C++MP will begin. Such a transfer will only start after a deliberate study of the commonalities and differences between the two sides. A research value of this paper is to provide such an insight. We anticipate that, when the C++MP problems that FP can solve are well-identified, new research threads will emerge for simulating the FP solutions in C++. Our other research aim here is to pave that way.

A note on laziness is relevant here: C++ templates are lazy from many viewpoints. For example, nested entities are left unevaluated until instantiation. This is the basis of how Sipos et al. [31] managed to implement compile-time infinite sequences in C++. However, as also alluded to by Sinkovics [28], although C++MP is a lazy FP language with selective strictness, eager template evaluation is often enforced predominantly. Namely, in practice, one can consider C++MP a strict FP language with selective laziness. So, as also shown throughout this paper, cross-lingual development between C++MP and the latter sort of languages (such as F# and Scala) works much better than HASKELL.

1.2 Scope

The Concept Check library is an old member of Boost with no enough use even in Boost itself. Furthermore, although rigorous studies [7, 14, 17] show the excess of commonalities between C++ Concepts and HASKELL Type Classes, the former failed to make it to the C++11 standard [2]. Recent endeavours on revisiting Concepts focus on categoric rewrite of the matter for Boost [6]. In current C++MP, constraints on types are usually enforced using the `mpl::if` family or `enable_if/disable_if` [13]. Describing more complicated constraints like Functional Dependencies needs more diligence (even in presence of Concepts) [8], hence they are not widely used. On the other hand, the recent proposals for the addition of a `static if` to C++ [5, 36] has raised interest in current C++MP. This is mainly because arbitrary compile-time expressions can form `static if` conditions.

All in all, despite their technical and theoretical benefits, Concepts have never received enough attention in the practise of C++MP. More to the point is that even with FP growing popularity in C++, and despite the rich record of Type Classes in FP, today there is no evidence in Concepts gaining more weight in C++MP. As a result, this paper neglects Concepts for it aims to facilitate C++MP as it is widely practised.

Given that no full implementation is ever provided for the entire semantics of any mainstream FP language, not much can be gained from comparing the informal semantics of C++MP against the (partly less) informal semantics of FP languages in a rigorous mathematical way. Instead, we believe comparing the languages in practice can be more constructive, especially once thinking about facilitating C++MP. Therefore, we do not follow [24] and [37] that carve formal specifications for parts of C++MP as a means, and, those papers fall out of scope for us.

We would finally remind that, by the time of this writing, C++11 is not yet widely-supported by compiler vendors. The features of C++11 which facilitate C++MP have, accordingly, not yet gained enough gravity in practice and we do not study them here.

1.3 Structure

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 so-called *0x features* of C++11. In particular, as opposed to the C++11 *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. The parts of our C++MP code which are not considered in the text can be found in Appendix A. 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 each 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 discussing related work in Section 2. We briefly explore the preliminaries in Section 3. Our technical work starts in Section 4 where we compare the FP nature of C++MP with the three FP languages. We then discuss how to represent $\mathbb{Q}$ in Section 5. Section 6.1 discusses how to reduce code repetition solely in C++MP. Next, Sections 6.2 and 6.3 discuss how to implement comparatives and arithmetics. Detailed discussion, finally, follows in the conclusion.

2. Related Work

Golodotz [10] offers a tour on the FP nature of C++MP by showing how to implement certain metaprograms by mimicking the respective HASKELL programs. Sipos et al. [31] 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 [23]. They advertise that their `Eval<>` metafunction evaluates the produced metaprograms according to the operational semantics of CLEAN. As also discussed in [27], whilst they do not formally present their operational semantics, their informal explanation suggests remarkable differences between the operational semantics of CLEAN and that of theirs.

CLEAN inherits the **(app)** of Launchbury for lazy evaluation [16]. Function application in the suggested operational semantics of Sipos et al. takes several forms as depicted in Figure 1. Their (@) is essentially the **(app)** rule of Launchbury. They also add ($\lightning$) for strict application. This mere addition makes equivalence of their operational semantics and that of CLEAN questionable. It is noteworthy that the operational semantics of van Eekelen and de Mol [33] suffers from increase of expressiveness upon evaluation of let-expressions [26]. Counter-intuitively enough, in such an operational semantics, $e \lightning x$ is not proven to be observationally equivalent to x `seq` $e\ x$. Finally, their **(bind)** rule is simply to optionally postpone function application.

Sinkovics [28] offers certain solutions for improving the FP support in Boost.MPL and discusses why they are needed. Sinkovics and Porkoláb [30] 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 [38] extension of their λ-library to full support for HASKELL. No complete release of their works is unfortunately available online for experimentation. We can therefore not evaluate their works any further. [2] Sinkovics [29] offers a restricted solution for emulating let-bindings in template metaprogramming.

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

$$\frac{\{f,x\} \subseteq dom(\Gamma)}{\Gamma : f\ x \Downarrow \Gamma : f\#x}\ (\textbf{bind}) \qquad \frac{\Gamma : f\ x \Downarrow \Delta : v}{\Gamma : f\#x \Downarrow \Delta : v}\ (\beta_@) \qquad \frac{\Gamma : f\!\downarrow x \Downarrow \Delta : v}{\Gamma : f\#x \Downarrow \Delta : v}\ (\beta_\downarrow)$$

$$\frac{\Gamma : f \Downarrow \Theta : \lambda y.e \quad \Theta : e[x/y] \Downarrow \Delta : v}{\Gamma : f\ x \Downarrow \Delta : v}\ (@)$$

$$\frac{\Gamma : f \Downarrow \Theta_1 : \lambda y.e \quad \Gamma : x \Downarrow \Theta_2 : v_x \quad \Theta_1 \bowtie \Theta_2 : e[v_x/y] \Downarrow \Delta : v}{\Gamma : f\!\downarrow x \Downarrow \Delta : v}\ (\downarrow)$$

Figure 1. Function Application of Sipos et al.

In our earlier work [27], we were the first to provide a real-world stand-alone exemplification of C++MP being Functional in nature. We explored impediments against fully-automatic cross-lingual development between C++MP and HASKELL. Armed with that, we suggested the approachability of a semi-automatic cross-lingual development between C++MP and hybrid FP languages.

Milewski [19] has a number of posts on his personal blog that speak about Monads [20], their benefits for C++, and how to implement Monadic entities in C++. In a later post, he explains how Monads in HASKELL can help the understanding of Boost.Proto – one of the most complicated C++MP libraries. 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 [25] shows how to implement algebraic datatypes in C++.

3. Preliminaries

Templates were originally to enable compile-time *genericity by type* [9, §2.2] in C++, which, in crude terms, is compile-time code reuse for every type. The `struct S` (line 2 below), for example, is meant to work for every type `T1` and `T2`. Likewise, function `f` (line 4 below) is meant to work for every type `T`:

```
1  template <typename T1, typename T2>
2  struct S {typedef ... type;};
3
4  template <typename T> void f(T) {...}
5
6  template<typename T> struct S<T, int> {...};
7  template<> void f(float) {...}
8
9  template<int n>
10 class C
11 {
12   typedef typename S<...>::type type
13 };
```

Templates can be specialised for types implementations of which may differ from the general one. (See the above lines 6 and 7 for the syntax.) Pattern matching is used to choose between the available implementations. Yet, C++ always chooses the best match in pattern matching, whereas in many FP languages including HASKELL, the earliest match is always chosen. Templatisation can be over compile-time integral constants too (such as class `C` above) for which specialisation is also allowed.

Templates can have nested types/values. For example, the structure `S` above defines `type` as a nested type. When a compiler fails to infer whether a token is a nested type or other sorts of nested entity, use of the keyword `typename` informs the compiler that a nested type is the intention. (See line 12 above for example.)

MPL [11] is the main metaprogramming component of the Boost C++ library. We use the following items from MPL, which we explain very briefly:

- `integral_c<T, n>` is a type representation for a compile-time constant `n` of integral type `T`.
- `bool_<b>` is a type representation for the compile-time Boolean constant `b`. `bool_<>` uses `true_` and `false_` as its compile-time equivalents for `true` and `false`.
- `not_<T>::type` is a type representing the Boolean complement of the one represented by `T`.
- `if_c<b, T1, T2>::type` is equivalent to `T1` if `b` is true, and to `T2` otherwise.

An FP language which does not allow side effects is called a *pure* FP language. When an argument is only evaluated if needed and the result of evaluation is shared thereafter in the scope, the argument is said to be evaluated *lazily* [16].

4. Greatest Common Divisor

Fraction cancellation is the cornerstone of any arithmetic on $\mathbb{Q}$ – that is, dividing the numerator 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 virtually *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 more verbose in this case – although not as verbose as the C++ one. We postpone the Scala version until Section 5 where we discuss how to represent the abstract datatype.

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

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 6.2, 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.

As a final remark, note that, due to the specific nature of *gcd*, the first line of code above is in fact optional in HASKELL. The second line would automatically perform the same job in the absence of the first. In C++, on the other hand, the second specialisation is necessary, or `GCD<0,0>` would be ambiguous.

5. Representing Rational Numbers

For a compile-time representation of $\mathbb{Q}$, the numerator 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 numerator 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 numerator and denominator, we use unsigned types for both. That entails storing the sign of a fraction in a separate Boolean template parameter (`negative`). The Scala version closely takes after its C++ counterpart:

```
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)

  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.

Had we chosen the numerator to carry the sign of the fraction by making it signed, the numerator's maximum absolute value was enforced to be half that of the denominator. Our original intention for this design was the pervasiveness of such considerations in C++MP. However, as seen above, this becomes significant in cross-lingual development between C++MP and Scala.

The fact that `gcd` is a private member function here is remarkable. With OOP being one of the major C++ paradigms, the same encapsulation is most likely to be practised in C++MP. Yet, we chose not to discuss that in Section 4 to stay focused.

The F# version is not as close to C++ as Scala. Local `let`-bindings are only internally available. Hence, getters `Num`, `Den`, and `Neg` are needed for outside queries. Furthermore, because function arguments cannot have default values in F#, one needs to add two constructor overloads which relegate the task to the main one (lines 9 and 10 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 6).

```
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 = num
  member this.Den = den
  member this.Neg = n

  override this.ToString() =
```

```
    if p <> 0u then
      (if this.Neg then "-" else "") +
      (sprintf "%i" this.Num) +
      (if this.Den = 1u then "" else "/" +
        sprintf "%i" this.Den)
    else
      "0"
```

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 9 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 = 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 numerators 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-numerator fraction is always written as "0" with no sign; when the denominator of a fraction is "1", only the numerator 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 order to automate cancellation, a HASKELL programmer might hide the `FractionH` constructor behind a function

```
fraction p q n = FractionH (p `div` g) (q `div` g) n
  where ...
```

in a module that only exports the function `fraction`. Firstly, this does not nicely map to C++MP. Secondly, it will not eliminate the on-the-fly cancellation and will, therefore, not facilitate C++MP at all.

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
```

6. 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.

6.1 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. C++ programmers evade similar manual implementations using preprocessor metaprogramming, as we explain next.

We start by providing the list of operation names (`RATIONAL_OPS`) that is constructed in exactly the same way as one employs *cons* in FP to construct lists.

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

The legwork is done using `BOOST_PP_LIST_FOR_EACH` that is, in fact, the preprocessor equivalent of the famous *map* function in FP.

```
BOOST_PP_LIST_FOR_EACH(OP_INIT_TEMPLATES,
                       type,
                       RATIONAL_OPS)
```

For each operation name n in `RATIONAL_OPS`, this generates a copy of `OP_INIT_TEMPLATES` in the body of which `OpName` is replaced by n and `NestedType` is replaced by `type`. The final piece of this puzzle is `OP_INIT_TEMPLATES`, which is defined below. Obviously, `OP_INIT_TEMPLATES` needs to be defined before its use by `BOOST_PP_LIST_FOR_EACH`, or the compiler might not find it.

```
#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;                       \
};
```

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 programmatically generates all the classes 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 7 for more.

6.2 Comparatives

The general idea to examine equality of two fractions is to examine the respective cancelled numerators 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.Neg),
        (f2.Num, f2.Den, f2.Neg) 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 the Scala keyword `case` in front of `FractionS` makes it a *case class*. The particular benefit of that, which is 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 given 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`, respectively.

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. [3]

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. See Appendix A for more.

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

A note on laziness seems suitable here: We could have chosen to implement the operations *lazily*, say using techniques presented in [3]. For example, we could have chosen `NotEqualTo` to be implemented as:

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

The benefit of this technique is that the mere instantiation of `NotEqualTo<T1, T2>` will not trigger the computation. Instead, `NotEqualTo<T1, T2>` can be freely passed around with the computation only taking place the first time that `NotEqualTo<T1, T2>::apply::type` is queried. Whether an automatic translation from HASKELL to C++ should by default choose the lazy metaprogramming or the strict one can be controversial. See Section 7 for more.

6.3 Arithmetics

Arithmetic operations on types are 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. Check Appendix A for the remaining operations.

```
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;
```

[3] Due to space constraints, we also skip the F# idiomatic override of `Equals` and `GetHashCode` that is used to update the standard collections with the respective canonical equality rules.

```
  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 operations. 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.

```
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 numerators 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 cancellation 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.

```
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, although not as close to C++ as Scala, is roughly of the same length. In F#, pattern matching on a *class* in need of similar construction-point normalisations and assertions is only emulated using queried fields. [4] But, pattern matching based on `Num`, `Den`, and `Neg` in one branch makes them equally accessible to other branch. Hence, the next branch uses the same constructs. This contrasts the C++ and Scala version where branches only query the constructs their respective part of algorithm entails.

```
let rec plus (f1: FractionF) (f2: FractionF) =
  match (f1.Num, f1.Den, f1.Neg),
        (f2.Num, f2.Den, f2.Neg) 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 = ...
```

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.

6.4 Samples

Complicated arithmetic expressions can now be evaluated using our libraries.

```
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;
```

[4] Note that our use of "class" here refers to the standard OOP piece of terminology. That aside, *active patterns* might address similar needs more naturally in F#. However, we will not consider them here for how they can perhaps facilitate C++MP is currently not entirely obvious.

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)
```

7. 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 misled earlier research towards suggesting automatic translation between FP languages and C++MP. Our earlier work demonstrated that to be unrealistic [27]. This work, however, suggests 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 – although more extensive research is still required.

One needs to bear in mind that this cross-lingual development does not eliminate every problem in C++MP. Instead, it is a proposal for making C++MP a two-phase (design/implementation) process. The outcoming benefit is reduction in the syntactic noise due to the fact that FP languages are by-design easier to do FP in – which, as we show in this paper, is the essence of C++MP too. Further details about the mechanics of such a semi-automatic cross-lingual development is subject for future research.

With this mindset, alongside the codes 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 closely similar across FP languages and C++MP. Figure 2 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 for the FP languages 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 2 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 [27], 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

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

○ = 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

Figure 2. Mechanical Translation between FP Languages and C++MP

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 2 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 a better option is perhaps a new front-end DSL 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.

Acknowledgements

We would like to thank Anton Tayanovskyy from IntelliFactory, and Miguel Garcia from EPFL for comments on the F# and Scala parts of our paper, respectively.

References

[1] *International Standard ISO/IEC 14882:1998(E): Programming Languages — C++* , (1998).

[2] *International Standard ISO/IEC 14882:2011: Information technology – Programming languages – C++*, (2011).

[3] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, AW Prof., 2004.

[4] 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.

[5] W. Brown, *A Preliminary Proposal for a Static `if`*, WG21/N3322 = PL22.16/12-0012.

[6] M. Calabrese, *Boost.generic: Concepts without concepts*, BoostCon 2011, http://boostcon.boost.org/program/sessions/.

[7] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, *Concepts: Linguistic Support for Generic Prog. in C++*, OOPSLA '06: Proc. 21st annual ACM SIGPLAN conf. Object-oriented prog. systems, langs., and applications (New York, NY, USA), ACM Press, 2006, pp. 291–310.

[8] D. Lincke and S. Schupp, *From HOT to COOL: Transforming Higher-Order Typed Languages to Concept-Constrained Object-Oriented Languages*, LDTA 2012: Proc. 12th Int. W. Lang. Descriptions, Tools, and Appl., 2012, pp. 17–30.

[9] J. Gibbons, *Datatype-Generic Prog.*, Spring School on Datatype-Generic Prog. (R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, eds.), LNCS, vol. 4719, Springer-Verlag, 2007, pp. 1–72.

[10] S. Golodotz, *Functional Programming Using C++ Templates (Part 1)*, Overload J. #81, http://accu.org/index.php/journals/1422.

[11] A. Gurtovoy and D. Abrahams, *The Boost C++ Meta-Prog. Library*, http://www.boost.org/.

[12] J. de Guzman, D. Marsden, and T. Schwinger, *Boost Fusion Library: Library for working with tuples, including various containers, algorithms, etc.*, http://www.boost.org/.

[13] A. Lumsdaine J. Järvi, J. Willcock and M. Calabrese, *Boost Ratio Library: Enable If*, http://www.boost.org/.

[14] J.P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz, *A Comparison of C++ Concepts and* HASKELL *Type Classes*, Proc. ACM SIGPLAN W. Generic Prog. (New York, NY, USA), WGP '08, ACM, 2008, pp. 37–48.

[15] V. Karvonen and P. Mensonides, *The Boost Library, Preprocessor Subset for C/C++*, http://www.boost.org/.

[16] J. Launchbury, *A natural semantics for lazy evaluation*, POPL '93: Proc. 20th ACM SIGPLAN-SIGACT Symp. Principles of Prog. Lang., ACM, 1993, pp. 144–154.

[17] M. Zalewski, A. Priesnitz, C. Ionescu, N. Botta, and S. Schupp, *Multi-language Library Development: From* HASKELL *Type Classes to C++ Concepts*, Multiparadigm Prog. with Object-Oriented Lang., an ECOOP W., 2007.

[18] Microsoft, *Rational Structure, Microsoft Solver Foundation 3.0*, http://msdn.microsoft.com/en-us/library/microsoft.solverfoundation.common.rational(v=vs.93).aspx#Y0.

[19] B. Milewski, *Bartosz Milewski's Prog. Cafe: Concurrency, Multicore, C++, Haskell*, http://bartoszmilewski.wordpress.com/.

[20] E. Moggi, *Notions of Computation and Monads*, Inf. & Comp. **93** (1991), no. 1, 55–92.

[21] E. Niebler, *Boost Proto Library: Expression template library and compiler construction toolkit for domain-specific embedded langs.*, http://www.boost.org/.

[22] P. Moore, *Boost Rational Library: A rational number class*, http://www.boost.org/.

[23] R. Plasmeijer and M. van Eekelen, *Concurrent* CLEAN *Language Report (v 2.0)*, Dec 2001, http://www.cs.kun.nl/~clean/contents/contents.html.

[24] R. Garcia and A. Lumsdaine, *Toward Foundations for Type-Reflective Metaprogramming*, ACM SIGPLAN Notices **45** (2010), no. 2, 25–34.

[25] D. Sankel, *Algebraic Data Types Series*, C++Next: The next generation of C++, http://cpp-next.com/archive/2010/07/algebraic-data-types/.

[26] S.H. HAERI, *Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness*, Proc. Int. Conf. Theoretical and Math. Foundations of Comp. Sci. (TMFCS-10), 2010.

[27] 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.

[28] 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.

[29] ———, *Nested Lamda Expressions with Let Expressions in C++ Template Metaprograms*, WGT'11 (Z. Porkoláb and N. Pataki, eds.), vol. III, Zolix, 2011, pp. 63–76.

[30] Á. Sinkovics and Z. Porkoláb, *Expressing C++ Template Metaprograms as Lambda Expressions*, TFP (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.

[31] 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.

[32] E. Unruh, *Prime Number Computation*, 1994, ANSI X3J16-94-0075/ISO WG21-462.

[33] M. van Eekelen and M. de Mol, *Mixed lazy/strict graph semantics*, Impl. and Appl. of Func. Langs., 16th Int. W., IFL04 (Lüebeck, Germany) (C. Grelck and F. Huch, eds.), Tech. Rep. 0408, Christian-Albrechts-Universität zu Kiel, September 2004, pp. 245–260.

[34] T. Veldhuizen, *Expression Templates*, C++ Report **7** (1995), no. 5, 26–31.

[35] V.J. Botet Escribá, *Boost Ratio Library: Compile time rational arithmetic*, http://www.boost.org/.

[36] W. Bright, H. Sutter, and A. Alexandrescu, *Proposal: `static if` declaration*, JTC1/SC22/WG21 N3329.

[37] W. Miao and J. Siek, *Incremental Type-Checking for Type-Reflective Metaprograms*, ACM SIGPLAN Notices **46** (2011), no. 2, 167–176.

[38] Z. Porkoláb and Á. Sinkovics, *C++ Template Metaprogramming with Embedded Haskell*, Proc. 8th Int. Conf. Generative Prog. & Component Engineering (GPCE 2009) (New York, NY, USA), ACM, 2009, pp. 99–108.

A. Our Remaining C++MP Code

In this section, we provide the parts of our C++MP code which are not discussed in the text. The techniques used here are all discussed enough in the previous sections, however.

```
template<typename T>
struct Negate
{
  typedef typename Negate<typename T::type>::type type;
};

template<long unsigned p, long unsigned q, bool n>
struct Negate<Rational<p, q, n> >
{
  typedef Rational<p, q, !n> type;
};

template<typename T>
struct Inverse
{
  typedef typename Inverse<typename T::type>::type type;
};

template<long unsigned p, long unsigned q, bool n>
struct Inverse<Rational<p, q, n> >
{
  typedef Rational<q, p, n> type;
};

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

template
<
  long unsigned p1, long unsigned q1,
  long unsigned p2, long unsigned q2, bool n
>
struct Less<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{
  typedef mpl::bool_
  <
   (p1 == 0 && p2 == 0)? false: n
  > type;
};

template
<
  long unsigned p1, long unsigned q1,
  long unsigned p2, long unsigned q2, bool n
>
struct Less<Rational<p1, q1, n>, Rational<p2, q2, n> >
{
  typedef typename Rational<p1, q1, n>::num_type
    num1_type;
  typedef typename Rational<p2, q2, n>::num_type
    num2_type;
  typedef typename Rational<p1, q1, n>::den_type
    den1_type;
  typedef typename Rational<p2, q2, n>::den_type
    den2_type;

  typedef typename mpl::bool_
  <
   (num1_type::value * den2_type::value <
    num2_type::value * den1_type::value)
  > raw_result;
  typedef typename mpl::if_c
  <
   n && NotEqualTo
        <
         Rational<p1, q1, n>, Rational<p2, q2, n>
        >::type::value,
   typename mpl::not_<raw_result>::type,
   raw_result
  >::type type;
};
```

```
template<typename T1, typename T2>
struct LessEqual
{
  typedef typename mpl::or_
  <
   typename Less<T1, T2>::type,
   typename EqualTo<T1, T2>::type
  >::type type;
};

template<typename T1, typename T2>
struct Greater
{
  typedef typename mpl::not_
  <
   typename LessEqual<T1, T2>::type
  >::type type;
};

template<typename T1, typename T2>
struct GreaterEqual
{
  typedef typename mpl::or_
  <
   typename Greater<T1, T2>::type,
   typename EqualTo<T1, T2>::type
  >::type type;
};

template
<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
>
struct Minus<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{
  typedef typename Plus
  <
   Rational<p1, q1, false>,
   Rational<p2, q2, false>
  >::type raw_result;
  typedef typename mpl::if_c
  <
   n,
   typename Negate<raw_result>::type,
   raw_result
  >::type type;
};

template
<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
>
struct Minus<Rational<p1, q1, n>, Rational<p2, q2, n> >
{
  typedef typename Rational<p1, q1, n>::num_type
    num1_type;
  typedef typename Rational<p2, q2, n>::num_type
    num2_type;
  typedef typename Rational<p1, q1, n>::den_type
    den1_type;
  typedef typename Rational<p2, q2, n>::den_type
    den2_type;

  typedef mpl::bool_
  <
   (num1_type::value * den2_type::value <
    num2_type::value * den1_type::value)
  > swap_needed;
  typedef typename mpl::if_
  <
   swap_needed,
   mpl::integral_c
   <
    long unsigned,
    num2_type::value * den1_type::value -
      num1_type::value * den2_type::value
   >,
   mpl::integral_c
   <
    long unsigned,
    num1_type::value * den2_type::value -
      num2_type::value * den1_type::value
   >
  >::type raw_num;
  typedef Rational
  <
   raw_num::value,
   den1_type::value * den2_type::value,
   n
  > raw_result;
  typedef typename mpl::if_<
   swap_needed,
   typename Negate<raw_result>::type,
   raw_result
  >::type type;
};

template
<
 long unsigned p1, long unsigned q1, bool n1,
 long unsigned p2, long unsigned q2, bool n2
>
struct Multiplies
<
 Rational<p1, q1, n1>,
 Rational<p2, q2, n2>
>
{
  typedef typename Rational<p1, q1, n1>::num_type
    num1_type;
  typedef typename Rational<p2, q2, n2>::num_type
    num2_type;
  typedef typename Rational<p1, q1, n1>::den_type
    den1_type;
  typedef typename Rational<p2, q2, n2>::den_type
    den2_type;

  typedef Rational
  <
   num1_type::value * num2_type::value,
   den1_type::value * den2_type::value,
   false
  > raw_result;
  typedef typename mpl::if_c
  <
   n1 == n2,
   raw_result,
   typename Negate<raw_result>::type
  >::type type;
};

template<typename T1, typename T2>
struct Divides
{
  typedef typename Multiplies
  <
   T1, typename Inverse<T2>::type
  >::type type;
};
```