

# C++ templates/traits versus Haskell type classes

Sunil Kothari \*  
Majoris Systems,  
Bangalore, India  
sunkot@yahoo.com

Martin Sulzmann  
School of Computing,  
3 Science Drive 2,  
National University of Singapore,  
Singapore  
sulzmann@comp.nus.edu.sg

October 31, 2004

## Abstract

This article presents an in-depth study of the close connection between Haskell type classes and C++ template/traits mechanism - two different facilities for implementing generic programming concepts. Haskell type classes and various extensions can be closely mimicked by C++ templates/traits and related mechanisms. We highlight the subtleties related to type-based computations and the limitations induced by language design on the program behaviour by a number of examples.

## 1 Introduction

There is a new thrust in programming languages community for language-level support for Generic Programming (writing code that works with any data type meeting a set of requirements) [GJL<sup>+</sup>03]. A recent survey [CDST04] of widely used programming languages has shown that Haskell<sup>1</sup> [JHA<sup>+</sup>99] SML [MTHM97] and C++ [Org98] come quite close in their support for generic programming. In particular, Haskell *type classes* [WB89] and C++ *templates/traits* [Mye95] mechanism provides the ability to express these requirements into the type system provided by the language.

Although type classes are well studied and have reached a relatively stable phase, C++ templates, on the other hand, are less structured (a collection of ad-hoc tricks) and, in some situations boil down to plain hackery. Given that C++ compiler is a Turing-complete interpreter for a subset of C++, theoretically, it is possible to emulate various type class extensions in C++ but there is not

---

\*This work was done by the author while on an internship with the second author at School of Computing

<sup>1</sup>Haskell98 with support for multi-parameter type class extension

much literature available that points to the precise connections. We show *how* to emulate Haskell type classes and related extensions in C++. We believe that the connection between the two has never been explored to such depths before.

The rest of this article is organized as follows: Section 2.1 introduces C++ templates/traits. Section 2.2 introduces Haskell type classes and other extensions. Section 2.3 gives a brief introduction on the theory behind modeling of functional dependencies as CHRs. Section 3 describes the close connection between Haskell type classes and C++ traits/templates. Finally, we conclude with Section 4 and list possible future work.

All C++ examples mentioned in this article have been tested on gcc version 3.3.6 running on Debian Linux. This is important as the support for templates, especially template template parameters, varies greatly with various gcc versions.

## 2 Background

This article assumes a preliminary knowledge of Haskell and C++ template specializations and some concept of meta-programming as related to C++ templates. A good reference on templates is [VJ02] and [CE00]. A good introductory article on template meta-programming is [GA], [Wal]. The treasure house for Haskell is inevitably the Haskell web-site [Com]. In the next few sub-sections we provide the relevant background on the rest of the concepts used.

### 2.1 C++ Templates/Traits

C++ traits technique was first mentioned in this article by Nathan C. Myers [Mye95]. Consider Myer's description of Traits:

A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that extra level of indirection that solves all software problems.

We will see later how “indirection” is achieved in Haskell by means of type class mechanism.

**Example 1.** *Consider this example from Myer's article. The problem is we want to define a class that encapsulates the behavior of reading an input stream. Normally, the stream is represented as a sequence of characters and EOF is defined as a value different from any of these character values. Traditionally, the type of EOF is `int` and the function that retrieves characters returns an `int`. The behavior is modeled as a C++ class below:*

```
class basic_streambuf {
    ....
    int sgetc(); // return the next character, or eof
    int sgetn(char*, int N) // get N characters.
};
```

Consider parameterizing `basic_streambuf` on the character type so as to handle a variety of different character types. This means that the type of `EOF` now becomes dependent on the type of characters. The parameterized version of the `basic_streambuf` class looks like this:

```
template <class charT, class intT>
class basic_streambuf {
    ....
    intT sgetc();
    int sgetn (charT*, int N);
}
```

And here lies the problem. There is nothing wrong as such (the program will compile gracefully) but this extra dependency is annoying. Moreover, given a value of `charT` there is only a particular value that corresponds to legal types. We would like to express this dependency in terms of program code in such a manner that the right type of return type of `sgetc` is selected on the basis of `charT`.

With traits it is easy. We can provide a default traits class template and specialize it for known character types.

```
template <class charT>
struct ios_char_traits { };
```

Note this default traits template is empty instead of providing some generic code. In a way it makes sense if the behavior is not defined for an arbitrary type then better state nothing about it. For `char` we specialize the template as:

```
struct ios_char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    static inline int_type eof() { return EOF; }
};
```

Similarly, we can add support for `w_char` (wide character) types:

```
struct ios_char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    static inline int_type eof() { return WEOF; }
};
```

Note here that we go from general to specific; first the generic code is defined and then specific instances. We have not seen the magic of traits yet. The magic happens in the way above traits are used:

```
template <class charT>
class basic_streambuf {
public:
```

```

typedef ios_char_traits<charT> traits_type;
typedef traits_type::int_type int_type;
int_type eof() { return traits_type::eof(); }
...
int_type sgetc();
int sgetn(charT*, int);
};

```

Here the “indirection” takes place in the first two lines after `public` keyword. That’s not all. In fact, a variant of the same program expressed below has similar semantics but instead of having a single type parameter it takes two type parameters; one expressed in terms of another. This demonstrates that we can not only define a general use of a specific type but also a specific use of a specific type.

```

template <class charT, class traits = ios_char_traits<charT> >
class basic_streambuf {
public:
    typedef traits traits_type;
    typedef traits_type::int_type int_type;
    int_type eof() { return traits_type::eof(); }
    ...
    int_type sgetc();
    int sgetn(charT*, int N);
};

```

□

Here’s another example of traits:

**Example 2.** Consider writing a library for numerical analysis domain. A library writer is not interested in various constants which are type dependent and already provided by the header file `float.h` i.e. parameters like mantissa, an epsilon etc. defined for `float`, `long`, `long double`. A template parameterized on the numeric type doesn’t know whether to refer to `FLT_EPSILON` or `DBL_EPSILON`. Again, traits solves the problem. Here is the code with template specializations handling the value for `float`, `double`:

```

template <class numT>
struct float_traits {};

template<>
struct float_traits<float> {
    typedef float float_type;
    static float_type epsilon() { return FLT_EPSILON; }
};

```

```

template<
struct float_traits<double> {
    typedef double float_type;
    static float_type epsilon() { return DBL_EPSILON; }
};

```

Now “epsilon” can be referred in a generic way as follows:

```

template <class numT >
class myclass {
public:
    typedef numT num_type;
    typedef float_traits<num_type> traits_type;
    num_type epsilon() { return traits_type::epsilon(); }
};

```

□

The usefulness of traits can be summarized as Myers describes it:

This technique turns out to be useful anywhere that a template must be applied to native types, or to any type for which you cannot add members as required for the template’s operations.

Although helpful in indirecting to a type-specific code, we can’t indirect code when the type is constrained by some complicated constraints. In other words, we don’t have the Haskell style of controlled overloading. We describe it next.

## 2.2 Haskell type classes

Haskell type classes were introduced in [WB89] to add controlled overloading of symbols to a Hindley-Milner based type system. Type classes allow the programmer to define relations over types. For single-parameter type classes, the type class relation simply states set membership. Consider the Eq type class, the declaration

```

class Eq a where
    (==):: a -> a -> Bool

```

states that every type `a` in type class `Eq` has an equality function “`==`”. Instance declarations prove that a type is in the class, by providing appropriate functions for the class methods. For example, `Int` is in `Eq`:

```

instance Eq Int where
    (==) = primeIntEq

```

which states that the equality function for `Ints` is `primeIntEq` where `primIntEq` is a built-in primitive function on Integers. The `==` function type can be *constrained* by allowing only values that have a type that is a member of class `Eq`:

```
(==)::Eq a => a -> a -> Bool
```

Multi-parameter type classes [Jon00] allow for multiple class parameters. For example,

```
class Collects e ce where
  empty:: ce
  insert :: e -> ce -> ce
```

The Collects type class defines a relation between element types `e` and the type `ce` of the collection itself. But the type `ce` of `empty` would lead to ambiguity as we cannot determine which instance to use for a given type of `ce`. Jones proposed *functional dependency* [Jon00] to make it unambiguous. The functional dependency `ce -> e` states that for all instance declaration of `Collects` the element type `e` can be determined from the collection type `ce`. We can now model a type class using the features that we have already discussed above.

**Example 3.** *Consider a type class that handles overloading of binary operators for `Int` and `Float`.*

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
instance Num Int where
  (+) = addInt
  (*) = mulInt
instance Num Float where
  (+) = addFloat
  (*) = mulFloat

square :: Num a => a -> a
square x = x * x
```

*This syntax hides the “indirection” that handles the overloading. This can be understood by looking at the translated version of the above example.*

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a)
add (NumDict a m) = a
mul (NumDict a m) = m
numDInt :: NumD Int
numDInt = NumDict addInt mulInt
numDFloat :: NumD Float
numDFloat = NumDict addFloat mulFloat
square' :: NumD a -> a -> a
square' numDa x = mul NumDa x x
```

□

The indirection is achieved by passing the appropriate dictionary at runtime. Since dictionaries are passed on the basis of instance types, any failure to disambiguate between the instances leads to compile-time errors.

## 2.3 CHRs and Functional Dependencies

Constraint handling rules [Fru95] (CHR) are a multi-headed concurrent constraint language for writing incremental constraint solvers. In effect, they define transitions from one constraint to an equivalent constraint. Transitions serve to simplify constraints and detect satisfiability and unsatisfiability. Constraint handling rules (CHR *rules*) are of two forms:

$$\begin{array}{ll} \text{**simplification**} & \text{rulename@ } c_1, \dots, c_n \iff d_1, \dots, d_n \\ \text{**propagation**} & \text{rulename@ } c_1, \dots, c_n \implies d_1, \dots, d_n \end{array}$$

In these rules  $c_1, \dots, c_n$  are CHR constraints and  $d_1, \dots, d_n$  are either CHR or Herbrand constraints. The simplification rule states that given a constraint set  $\{c_1, \dots, c_n\}$  this set can be replaced by  $\{d_1, \dots, d_n\}$ . The propagation rule states that given a constraint set  $\{c_1, \dots, c_n\}$  we should add  $\{d_1, \dots, d_n\}$ . A general framework for type class and various extensions has been proposed in [GSS01] [GSS00]. We use CHR rules to model functional dependencies in multi-parameter type classes. Consider our earlier example which models a collection reproduced below:

```
class Collects e ce | ce -> e
```

The functional dependency `ce -> e` can be expressed in terms of CHRs as:

$$\begin{array}{l} \text{Collects } e \text{ ce}, \text{ Collects } f \text{ ce} \implies f = e \\ \text{Collects } [a] \text{ b} \implies a = b \end{array}$$

The first rule states that if we have two constraints (`Collects e ce`) and (`Collects f ce`) both hold then have that `e` and `f` are of the same type. The second rule states that if (`Collects [a] b`) holds then it must be that `a` is of the same type as `b`.

There are other CHRs for handling constraints that model class superclass relationship and the corresponding instances but for our purpose the above two rules are sufficient. Interested readers can refer [GSS01] for a formal theory behind CHRs and type classes. In the next sub-section we detail an important principle that provides a way to emulate Haskell type class instances in C++.

## 2.4 SFINAE principle

In C++, a template function call resolution is a two-step process. First, the argument type is deduced from the function call and then the argument type is substituted in the function templates to resolve any overloads. It is at this stage of overloading that if an incorrect argument type is formed the template is removed from the overload set without causing a compile-time error. This is called as substitution-failure-is-not-an-error (SFINAE) principle. There are various conditions where argument-types may be invalid. More details can be found in [JWHL03] ,[VJ02]. The following example demonstrates this:

**Example 4.** Consider this example taken from [JWL03] where we overload `negate` function.

```
int negate (int i)
{ return -i; }
```

Assume there is another definition of `negate` and this time its a function template:

```
template<class T>
typename T::result_type negate(const T & t){ return -t; };

int main()
{
    int i;
    i = negate(5);
    return 0;
}
```

A call to `negate` is handled by the overload resolution by considering both these definitions. The template version is instantiated as :

```
int::result_type negate(const int &);
```

The return type `int::result_type` is invalid since there are no nested types for `int`. Therefore, the template version is removed from the overload resolution set and the function call is resolved to the non-template version of `negate`.

□

Without SFINAE both the calls would be a candidate for the call and hence the program would fail to compile.

## 2.5 Enable If technique

Although the SFINAE principle in itself is relatively of little use but when combined with compile-time meta-programming it packs that extra punch that not only helps us to emulate type classes but implement various type-class extensions and gives new insights into Haskell type class and C++ templates/traits connection. We detail the principle here but use it later to emulate various advanced Haskell type class extensions.

**Example 5.** Consider a somewhat simpler form of `Show` defined in prelude

```
class Showable t where
    show :: t -> String
instance Showable Int where
    show i = ...
instance Showable t => Showable [t] where
    show l = ...
```

Similarly, we can define a `print` method that can take as input arguments only those types that are members of `Showable` type class.

```
print:: Showable t => t -> IO()
print s
```

□

To emulate this in C++, we use `enable_if` template [JWL03] to constrain a templates arguments. This in turn helps to enable or disable a particular class template specialization. The following example shows the main concepts involved:

**Example 6.** Consider this example where we encode `Showable` in C++.

```
template <class T, class Enable = void>
struct showable_traits { static const bool conforms = false;};
template<class T>
struct showable
{
    static const bool b = showable_traits<T>::conforms;
    if (b == false )
    {
        std::cout << ‘‘This type does not model showable concept’’ ;
    } else
    {
        showable_traits<T>::show();
    }
};
```

The first template class encodes whether a type `T` (any arbitrary type) is an instance of the type class; the boolean parameter `conforms` indicates whether a particular type participates in the concept being modeled. The default case is that type does not model a concept. To “enable” a type to participate in the concept being modeled the programmer needs to explicitly state that the parameter `conforms` is assigned the value `true`.

```
template<>
struct showable_traits<int> {
    static const bool conforms = true;
    string show(int x) /*...*/
};
```

This template specialization indicates `int` is now assigned to model the concept. Compare this with Haskell, where we need to explicitly state that a value of type `Int` is showable. The flexibility with type classes is that we can define additional constraints on an arbitrary set of types. In contrast, C++ we can make only a certain set of types showable. The following example makes all the pointer types showable:

```

template<class T>
struct showable_traits<T*> >{
    static const bool conforms = true;
    string show(T* x){ /*... */ }
}

```

Consider again the `Showable` type class. In Haskell, we can assign a context to a type declaration to specify for an arbitrary type `T` in the form of context as:

```
instance Showable T => Showable [T] where ....
```

Something similar can be expressed in C++ by using `enable_if` template:

```

template<class T>
struct showable_traits< list<T>,
                      typename enable_if<showable_traits<T>::conforms
                                      >::type
                      >
{
    static const bool conforms = true;
    string show(const list<T>& x) { ... }
};

```

□

Here's a summary of `enable_if` template as mentioned in [JWL03]. A class template specialization can be augmented with a static metaprogram that examines certain properties of the template arguments and conditionally enables or disables the specialization. This allows us to go beyond the capabilities of Haskell type classes. The `enable_if` technique can be used to express almost arbitrary boolean expressions which are evaluated at compile-time.

### 3 Emulating type classes in C++

In this section we show how we can mimic Haskell type classes and then show that with the help of some additional techniques we can easily emulate advanced type class behavior such as type classes with duplicate instances and multi-parameter type classes with functional dependencies. We start with looking at some sample programs to bring out the subtleties between Haskell type class and C++ templates.

#### 3.1 Type annotations

Almost similar behavior can be expressed in terms of Haskell type classes as follows:

**Example 7.** Consider the `stream.buf` class mentioned earlier in Section 2.1.

```
class StreamBuf a b where
  eof :: b
  sgetc :: b
  sgetn :: a -> Int -> Int
instance StreamBuf Char Int where
  eof = EOF
...
instance StreamBuf W_Char Int where
  eof = WEOF
...
```

□

Currently, Haskell supports only default type annotations and not *partial* type annotation. This would be an easy extension. In Haskell, the specific `instance` models the code to be executed at runtime based on the types provided at compile-time.

### 3.2 Handling structural conformance

**Example 8.** Consider another example taken from Stroustrup's book [Str93] and also finds a mention in [Mye95].

```
template <class T> class CMP {
  static bool eq(T a, T b) { return a == b; }
  static bool lt(T a, T b) { return a < b; }
};
```

and an ordinary string template:

```
template <class charT> class basic_string {
  // .....
};
typedef basic_string<char> string;
```

We can now define a function `compare()` that can be used to compare strings. More precisely, the criteria of sorting is now parameterized as shown in the following code:

```
template <class charT, class C = CMP<charT> >
int compare( const basic_string<charT>& str1,
             const basic_string<charT>& str2
            ) {
  for (int i=0; i<str1.length() && i < str2.length(); i++) {
    if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]);
  }
  return str2.length() - str1.length();
};
```

We can now pass our comparison criteria as an instantiated template class:

```
class LITERATE {
    static int eq(char a , char b) return a == b;
    static int lt(char, char); // use literary convention; defined elsewhere
};
string swede1, swede2;
compare<char, LITERATE>(swede1,swede2);
```

□

In Haskell, we can express the above example as:

**Example 9.** Consider this example where type signatures enforce constraints on implementations.

```
class CMP t where
    eq :: t -> t -> Bool
    lt :: t -> t -> Bool
instance CMP Char where
    eq = ..
    lt = ..

class CMP t => Compare [t] where
    compare :: [t] -> [t] -> Bool
```

□

In C++, something similar is expressed by encoding the parameter as an instantiated template but we do not force the structure requirement strictly. For example, any arbitrary class Foo can be passed as a parameter to the compare function template.

### 3.3 Overloading functions

In Haskell, we can describe a generic family of overloaded functions via instance declarations.

**Example 10.** Consider this example where we evaluate a binary node corresponding to a parse tree.

```
data BNode a b c = BNode a b c
data L = L
data R = R
data OpPlus = OpPlus
class Eval a where
```

```

evalAt :: a -> ()
instance (Eval L, Eval R) => Eval (BNode OpPlus L R) where
  evalAt(BNode OpPlus L R) = ... eval L; ... eval R; ...

```

□

Something similar can be expressed in C++ as is mentioned in [CDST04]

```

template <class L, class R> struct Eval<BNode<OpPlus,L,R> >
{ static inline T evalAt(const BNode<OpPlus, L,R>& b, int i)
  { return ... Eval<L>::evalAt ....Eval<R>::evalAt .....

```

In Haskell, we specify exactly the valid relations among overloaded instances whereas in C++ we annotate the program text (e.g. `Eval<L>::evalAt`), similar to System-F style type application. The `enable_if` technique can be used to express almost arbitrary boolean expressions which are evaluated at compile-time. This allows us to go beyond the capabilities of Haskell type classes. We cite below some important cases with examples.

### 3.4 Overlapping Instances

Our discussion until now has been focussed on mimicking Haskell type classes in C++.

**Example 11.** *Consider the following example in Haskell that involves overlapping instances.*

```

class C a b where
  f :: a -> b -> Bool
instance C Bool a where
  f b x = b
instance C a Bool where
  f x b = b

--Compiling Main                ( overlap.hs, interpreted )
--
--overlap.hs:4:
--  Overlapping instance declarations:
--    overlap.hs:4: C Bool a
--    overlap.hs:7: C a Bool
--Failed, modules loaded: none.

```

□

GHC fails to compile the above program. A similar program in C++ also fails to compile because of the ambiguity. Note here that we do not use any `enable_if` templates.

**Example 12.** Consider the above example in C++:

```

template<template <class T1, class T2 > class C, class T >
struct C_traits<C<T, int> >
{
    static const bool conforms = true;
    ....
};

template<template<class T1, class T2> class C, class T>
struct C_traits<C<int, T> >
{
    static const bool conforms = true;
    .....
};

```

Instantiating `C_traits<C<int,int> >` leads to the following error message in C++:

```

Hoverlappaper.cpp: In function 'int main()':
Hoverlappaper.cpp:285: error: ambiguous class template
instantiation for 'struct C_traits<C<int, int>, void>'
Hoverlappaper.cpp:259: error: candidates are: struct C_traits<C<int, T>, void>
Hoverlappaper.cpp:249: error:                 struct C_traits<C<T, int>, void>
Hoverlappaper.cpp:285: error: aggregate 'C_traits<C<int, int>, void> a' has
incomplete type and cannot be defined

```

These are essentially the same error messages as given for overlapping instances in Haskell. Note that no amount of compile-time meta-programming can disambiguate the above instances. But in certain cases, we can disambiguate between these overlapping instances. Consider a somewhat similar but a more general case where we would like to model the subtype relation  $\leq$  between regular expression types  $r1$ ,  $r2$  and  $r3$  given by:

$$\frac{True}{r1 \leq r1} \quad (RESubtype0)$$

$$\frac{r1 \leq r2}{r1 \leq (r2 \mid r3)} \quad (RESubtype1)$$

$$\frac{r1 \leq r3}{r1 \leq (r2 \mid r3)} \quad (RESubtype2)$$

**Example 13.** Consider this example in Haskell which models the above subtype relation.

```

data OR a b = data OR a b
class RESubtype r1 r2 where
instance RESubtype r1 r1
instance RESubtype r1 r2 => instance RESubtype r1 (OR r2 r3)

```

```
instance RESubtype r1 r3 => instance RESubtype r1 (OR r2 r3)
```

□

Again GHC fails to compile the above program. Clearly, in this example GHC issues warnings that the instances are duplicated even though the contexts disambiguate the two instances. This is where the `enable_if` technique proves so useful. As in the earlier examples we assume the existence of a default `Enabler` template and a default `RESubtype` template. The context can now be used to disambiguate between the two instances as shown below:

**Example 14.** *Consider this example where subtyping is encoded in C++.*

```
template< template<class T1, class T4> class RESubtype,
          template<class T1, class T2> class OR,
          class T3, class T2 , class T1
        >
struct RESubtype_traits< RESubtype< T1, OR<T2,T3> >,
                       typename enable_if<RESubtype_traits<RESubtype<T1,T3>
                                           >::conforms
                                           >::type
                       >
{
    static const bool conforms = true;
    static void equal(){
        std::cout <<"RESubtype<T1,T3> => RESubtype<T1,(T2|T3)>" << std::endl;
    }
};

template< template<class T1, class T2> class RESubtype,
          template<class T1, class T2> class OR,
          class T3, class T2 , class T1
        >
struct RESubtype_traits< RESubtype< T1, OR<T2,T3> >,
                       typename enable_if<RESubtype_traits<RESubtype<T1,T2>
                                           >::conforms
                                           >::type
                       >
{
    static const bool conforms = true;
    static void equal()
    {
        std::cout <<"RESubtype<T1,T2> => RESubtype<T1,(T2|T3)>" << std::endl;
    }
};
```

```
}  
};
```

□

The interesting part in this code is the place where `enable_if` is used to disambiguate between the instances. Thus, a call like `RESubtype<int,OR<int,float>>>::equal()` is resolved to the correct `equal` function. In our case, `equal()` outputs:

```
RESubtype<T1,T2> => RESubtype<T1,(T2|T3)>
```

Here we have assumed that there is code which rules out instances of the form `RESubtype<int,float>` but accepts instances of the form `RESubtype<int,int>`.

### 3.5 Functional Dependencies

So far we have encoded Haskell type classes which do not involve any functional dependencies.

**Example 14.** *Consider a simple example where functional dependencies are used to rule out illegal instances:*

```
class Foo a b | a -> b where ...  
instance Foo Bool Char  
instance Foo Bool Int
```

□

This program is ambiguous since for a given type of `a` we have two possible types for `b`. This can also be realized in terms of CHRs as:

$$Foo\ a\ b\ ,\ Foo\ a\ c\ \Longrightarrow\ b\ =\ c$$

Clearly, the above two instances conflict with what functional dependencies say about the type `a` and `b` i.e. `a` determines `b`.

**Example 15.** *Consider the following example which tries to encode functional dependencies in terms of C++ template code. The idea behind this code is that the two overload instances with the same first parameter will produce a compile-time error. We assume that we have the corresponding `enable_if` template defined.*

```

template< template<class T1, class T2> class Foo,
          class T1, class T2
          >
struct Foo_traits< Foo<T1,T2>,
                 typename enable_if<Foo<T1,_>::conforms>::type
                 >
{
    static bool const conforms = true;
    .....
};

template<>
struct Foo_traits<Foo<int,char> >
{
    static bool const conforms = true;
    .....
};

```

□

The above code will compile if and only if the overload resolution set finds only one element in the overload resolution set. **We need an extension of `enable_if` or maybe a new technique `optional_if` to handle such cases.** Assuming that we have such an extension we can then express more complex functional dependency. The following example elaborates this point further:

**Example 16.** *Consider earlier example but with both parameters dependent on each other:*

```
class Foo a b | a -> b, b -> a where ...
```

The corresponding C++ code remains the same but we make changes in the `enable_if` template to reflect the additional constraints.

```

template< template<class T1, class T2> class Foo,
          class T1,
          class T2
          >
struct Foo_traits< Foo<T1,T2>,
                 typename enable_if<Foo_traits<Foo<T1,_> >::conforms &&
                                     Foo_traits<Foo<_,T2> >::conforms
                                     >::type
                 >
{
    .....
}

```

□

This example correctly mimics the functionality of functional dependencies but it does not scale well when we encounter `Foo [a] b`. Because such a case involves type equivalence at two levels: at the level of types it enforces structural equivalence and at the level of values it enforces that values are uniquely determined. In such a case we should infer that `b = [c]` for some `c`. In C++, we can handle such a situation if we can inspect the types of `b` and `a` at compile-time and take a decision based on this information. We show how to do this next.

### 3.6 Type traits and functional dependencies

In C++, there is a mechanism by which it is possible to inspect the type structure at compile-time [MC00]. In our case, we can check for structural equivalence of a type; check whether the first parameter is of type list and the second parameter is of base type. An example follows:

**Example 17.** *Consider this example where we can find at compile-time whether a type `T` is a pointer type.*

```
template<typename T>
struct is_pointer
{ static const bool value = false; };
```

```
template<typename T>
struct is_pointer<T*>
{static const bool value = true; };
```

□

In C++, we can do more than just inspect types at compile-time. In fact, we can perform type-specific transformation; for example, we can remove a top-level qualifier `list` from a type `list<T>` as shown in the following example:

**Example 18.** *Consider this example where we use compile-time computation to remove outer list constructor `list` from a type `list<T>`.*

```
template<typename T>
struct removelist<std::list<T> >
{
    typedef T type;
};
```

```
template<typename T>
struct removelist<std::list<std::list<T> > >
{
    typedef std::list<T> type;
};
```

□

The above example removes the outer list constructor from the type  $T$ . For example, `removelist<list<int>>>::type` would evaluate to the type `int` whereas `removelist<list<list<int>>>>::type` would evaluate to the type `list<int>`.

This technique can also be used to add a list constructor to types. Thus, `addlist<list<int>>>::type` would evaluate to the type `list<list<int>>>` where `addlist` is defined just as `removelist`. In terms of CHRs the above example can be seen as:

$$F [a] b \implies F [a] b, b = [c]$$

The next sub-section shows an example involving above CHR encoded in C++.

### 3.7 Malicious Functional Dependency example

We now have a mechanism by which we can emulate functional dependencies in C++. But functional dependencies have been known to cause non-termination of type inference procedure. The following Haskell example shows what can go wrong with functional dependencies. Related information on the theory behind this example can be found in [DPJSS04].

**Example 19.** Consider this contrived example which involves a functional dependency.

```
class Foo a b | a -> b
instance Foo a b => Foo [a] [b]
```

□

This boils down to following rules in terms of CHR solving as:

$$\begin{aligned} \textcircled{\text{Rule1}} \quad & Foo\ a\ b, Foo\ a\ c \implies b = c \\ \textcircled{\text{Rule2}} \quad & Foo\ [a]\ c \iff c = [b], Foo\ [a]\ c \\ \textcircled{\text{Rule3}} \quad & Foo\ [a]\ [b] \implies Foo\ a\ b \end{aligned}$$

Now consider `Foo [a] a`,

$$\begin{aligned} \iff_{rule2} \quad & Foo\ [a]\ a, a = [b] \\ \iff \quad & Foo\ [[b]]\ [b], a = [b] \\ \implies_{rule3} \quad & Foo\ [b]\ b, a = [b] \\ & \dots \end{aligned}$$

We can code almost similar program behaviour in C++ as:



```

zipall _ _ = []

class Zip a b c | a c -> b , b c -> a where
    myzip :: a -> b -> c
instance Zip [a] [b] [(a,b)] where
    myzip a b = zipall a b
instance Zip [(a,b)] [d] e => Zip [a] [b] ([d]-> e) where
    myzip a b = \c -> (myzip (zipall a b) c)

e1 :: [(((Int,Char),Int),Char),Char]
e1 = myzip [1::Int,2] ['a','c'] [5::Int] ['x','y'] ['r','t']
--here is the output
--[(((1,'a'),5),'x'),'r']

```

The Haskell code above needs explicit type annotation and functional dependencies for type inference to guide the computation. The type class mentioned above defines a function that can accept *variable* number of arguments in a type-safe manner.

In C++, something similar can be expressed with the help of functor objects by overloading the () operator. We use a technique first described in [Ale98] for a container that can accumulate a variable number of arguments in a type-safe manner. Since the generic list container provided by STL is homogeneous we hold pointers to the base class instead of derived class as:

```

std::list<Base*> v1;
class Base
{
public:
    ...
};

```

To give a truly polymorphic behavior, we add wrapper classes for representing primitive types i.e.int, char:

```

class SimpleInteger:public Base;
std::list<Base*> v1;
SimpleInteger* si1 = new SimpleInteger(12);
v1.push_back(si1);

```

Class `SimpleInteger` acts as a wrapper class for integer type. All the wrapper classes inherit from the `Base` class in order to pass a value corresponding to the base type as pointers to this class type. But the bulk of the work is centered around `mycontainer` class which inherits from the `list` container. We highlight the skeleton of the equivalent program. The full code listing can be found in the Appendix A.

```

template < class T = Base, class container = std::list<T*> >
class mycontainer : public container

```

```

{
private:
    std::list<Base*> fst;
    std::list<Base*> snd;
public:
    mycontainer()
    {
    }

    mycontainer(const std::list<Base*> &v)
    {
        this->swap(v);
    }

    explicit mycontainer(std::list<Base*>& a, std::list<Base*>& b)
    {
        std::list<Base*> final;
        std::list<Base*>::iterator iter1;
        for(iter1 = a.begin(), iter2 = b.begin();
            iter1 != a.end() && iter2 != b.end();
            iter1++, iter2++)
        )
        {
            Pair* p1 = new Pair(*iter2, *iter1);
final.push_back(p1);
        }
        this->swap(final);
    }

    mycontainer& operator()( std::list<Base*> &a)
    {
        std::list<Base*>::iterator iter1, iter2;
        std::list<Base*> temp;
        for(iter1 = a.begin(), iter2 = this->begin();
            iter1 != a.end() && iter2 != this->end();
            iter1++, iter2++)
        )
        {
            Pair* p2 =new Pair(*iter2,*iter1);
temp.push_back(p2);
        }
        this->swap(temp);
        return *this;
    }
}

```

```
};
```

A call to the function `foo` ( which has the functionality of `zip` the Haskell equivalent) can now be specified as:

```
foo(make_list<Base>(v1,v2)(v3)(v4)(v5));
```

where `v1`, `v2`, `v3`, `v4` and `v5` denote a list of pointer to base type. The `make_list` function returns an instance of `mycontainer` and subsequent calls are operator overloads of `()` with a list of pointers to base type passed as argument. Here's the output of the above function call with the above arguments::

```
Pair<Pair<Pair<Pair<13,12>,13>,S>,k>  
Pair<Pair<Pair<Pair<13,12>,18>,U>,e>  
Pair<Pair<Pair<Pair<12,12>,13>,N>,n>
```

Here `Pair<>` is defined as a placeholder for holding two base pointers:

```
struct Pair:public Base  
{  
    Base* fst;  
    Base* snd;  
};
```

Our findings can be summarized as:

- Template/traits are a popular mechanism for type/value computations in C++.
- Haskell type classes and other extensions can be mimicked in C++ using some tricks (`enable_if`) and type traits technique.
- In Haskell, there is a clear phase distinction between type and value computations. Via type class we can impose properties on types. This information is then used to generate code by turning into dictionaries.
- In C++, type inference/checking is viewed as program manipulation whereas Haskell has type inference. Therefore, in C++ we need to annotate the program text (e.g. `Eval<L>::evalAt` ) (and also the fact that compile-time computation is based on the fact that values are immutable, we encounter “functional” view of type inference) whereas in Haskell we specify exactly the valid relations among overloaded instances. The `enable_if` technique can be viewed as a tool to model these instance relationships in C++.
- In C++, SFINAE principle along with the `is` is used to disambiguate between the duplicate instances. Correspondingly, Haskell lacks a mechanism whereby duplicate instances can be disambiguated at compile-time by using the context information.

- We think it is possible to encode in C++ the following example in Haskell where functional dependencies are inherited from the parent class. Here is an example:

```
class Foo a b | a -> b where ....
class Foo a b => Bar a b where ....
```

## 4 Conclusions

Haskell type classes and various extensions can be emulated in C++ by C++ templates/traits and some amount of compile-time metaprogramming. C++ is wordy and thus requires many more lines of codes than something similar in Haskell which can be expressed in few lines of code. Nevertheless C++ can easily handle other possible extensions to Haskell type classes. We believe that more work needs to be done to bring out the connections and potential problems associated with various language mechanisms. Our work will benefit language designers who might want ad-hoc polymorphism in a language that has no in-built support for controlled polymorphism( like type classes in Haskell) but otherwise provides excellent facilities for type-based computations (like Template meta-programming) and some facility for Generic Programming (like templates/traits) .

## References

- [Ale98] Andrei Alexandrescu. Inline Containers for Variable Arguments. *C++ Users Journal*, September,1998.
- [CDST04] Krzysztof Czarnecki, John Donnell, Joerg Striegnitz, and Walid Taha. DSL implementation in Metaocaml, Template Haskell, and C++. In *Not yet known*, volume 3016 of *LNCS*. Springer Verlag, 2004.
- [CE00] K. Czarnecki and U. Eisenecker. Generative programming: Methods, techniques, and applications, 2000.
- [Com] Haskell Community. Haskell web site. <http://www.haskell.org>.
- [DPJSS04] Gregory J. Duck, Simon Peyton-Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies. *ESOP 2004: The European Symposium on Programming*, 2004.
- [Fru95] Thom Fruehwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer Verlag, 1995.

- [GA] Aleksey Gurtovoy and David Abrahams. The boost c++ metaprogramming library. <http://www.boost.org/libs/mpl/doc/paper/html/index.html>.
- [GJL<sup>+</sup>03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 115–134. ACM Press, 2003.
- [GSS00] Kevin Glynn, Martin Sulzmann, and Peter J. Stuckey. Type classes and constraint handling rules. *First Workshop on Rule-Based Constraint Reasoning and Programming*, July 2000.
- [GSS01] K. Glynn, P.J. Stuckey, and M. Sulzmann. A General Type Class Framework. Technical report, Department of Computer Science, The University of Melbourne, 2001.
- [JHA<sup>+</sup>99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, and Brian Boutel. Haskell98: A Non-strict, Purely Functional Language, February 1999.
- [Jon00] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag, 2000.
- [JWHL03] J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.
- [JWL03] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In Frank Pfennig and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 228–244. Springer Verlag, September 2003.
- [MC00] John Maddock and Steve Cleary. C++ type traits. [http://www.boost.org/libs/type\\_traits/c++\\_type\\_traits.htm](http://www.boost.org/libs/type_traits/c++_type_traits.htm), 2000.
- [MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (Revised)*. MIT press, Cambridge, MA, 1997.
- [Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7:32–35, 1995.
- [Org98] International Standardization Organization. ANSI/ISO standard 14882, Programming Language C++. 1 rue de Varembe, Case postale 56, CH-1211 Geneve 20, Switzerland, 1998.

- [Str93] B. Stroustrup. *Design and Evolution of C++*. Addison Wesley, Reading, MA,USA, 1993.
- [VJ02] David Vandevorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [Wal] Josh Walker. Template metaprogramming: make your compiler work for you. <http://home.earthlink.net/~joshwalker1/writing/TemplateMetaprogramming.%html>.
- [WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. *ACM Symposium on Principles of Programming Languages*, 1989.

## Appendix A

```
// This is the original program which works
#include <vector>
#include <list>
#include <deque>
#include <iostream>

class Base
{

public:
    virtual ~Base(){}
    virtual void print(){}
};

class SimpleInteger:public Base
{
    int i;

public:
    SimpleInteger(int sint)
    {
        i = sint;
    }
    ~SimpleInteger(){
    }
    void print(){
        std::cout<<this->i;
    }
};
```

```

class SimpleCharacter:public Base
{
private:
    char c;
public:
    SimpleCharacter(char cchar)
    {
        c=cchar;
    }
    ~SimpleCharacter(){
    }
    void print(){
        std::cout<< this ->c << ", ";
    }
};

struct Pair:public Base
{
    Base* fst;
    Base* snd;

    Pair(Base* ffst,Base*  ssnd)
    {
        fst = ffst;
        snd = ssnd;
        //std::cout<< "Constructor 2 called"<< std::endl;
        //ffst->print();
    }
    void print()
    {
        std::cout<<"Pair<";
        fst->print();
        std::cout<<",";
        snd->print();
        std::cout<<">";
    }
    virtual ~Pair(){
};

template < class T = Base, class container = std::list<T*> >
class mycontainer : public container
{
private:

```

```

    std::list<Base*> fst;
    std::list<Base*> snd;
public:
    mycontainer()
    {
        std::cout <<"calling default constructor"<<std::endl;
    }

    mycontainer(const std::list<Base*> &v)
    {

        this->swap(v);
    }

    explicit mycontainer(std::list<Base*>& a, std::list<Base*>& b)
        //      : container(1, new Pair<T,T>(a,b))
    {
        std::list<Base*> final;
        std::list<Base*>::iterator iter1;
        std::list<Base*>::iterator iter2;
        for( iter1 = a.begin(), iter2 = b.begin();iter1 != a.end() &&iter2 != b.end();iter
        {

            Pair* p1 = new Pair((*iter2),(*iter1));
final.push_back(p1);

        }
        this->swap(final);
    }

    mycontainer& operator()( std::list<Base*> &a)
    {
        std::list<Base*>::iterator iter1, iter2;
        std::list<Base*> temp;
        for( iter1 = a.begin(), iter2 = this->begin();iter1!= a.end()&&iter2!= this->end()
        {

            Pair* p2 =new Pair(*iter2,*iter1);
temp.push_back(p2);

        }
        this->swap(temp);
        return *this;
    }
}

```

```

};

template<class T>
inline mycontainer<T> make_list(std::list<Base*>& a, std::list<Base*>& b )
{
    return mycontainer<T>(a,b);
}

template <class container>
void foo( container a)
{
    std::list<Base*>::iterator aiter;
    for(aiter = a.begin(); aiter !=a.end(); aiter++)
    {
        ((*aiter)).print();
        std::cout << std::endl;
    };
}

int main()
{
    std::list<Base*> v1;
    std::list<Base*> v2;
    std::list<Base*> v3;
    std::list<Base*> v4,v5;

    SimpleInteger* si1 = new SimpleInteger(12);
    SimpleInteger* si2 = new SimpleInteger(12);
    SimpleInteger* si3 = new SimpleInteger(12);
    SimpleInteger* si4 = new SimpleInteger(13);
    SimpleInteger* si5 = new SimpleInteger(13);
    SimpleInteger* si6 = new SimpleInteger(13);
    SimpleInteger* si7 = new SimpleInteger(18);
    SimpleCharacter* sc1 = new SimpleCharacter('S');
    SimpleCharacter* sc2 = new SimpleCharacter('U');
    SimpleCharacter* sc3 = new SimpleCharacter('N');
    SimpleCharacter* sc4 = new SimpleCharacter('I');
    SimpleCharacter* sc11 = new SimpleCharacter('k');
    SimpleCharacter* sc12 = new SimpleCharacter('e');
    SimpleCharacter* sc13 = new SimpleCharacter('n');
    SimpleCharacter* sc14 = new SimpleCharacter('n');
    v1.push_back(si1);
    v1.push_back(si2);

```

```
v1.push_back(si3);
v2.push_back(si4);
v2.push_back(si5);
v2.push_back(si3);
v3.push_back(si6);
v3.push_back(si7);
v3.push_back(si4);
v4.push_back(sc1);
v4.push_back(sc2);
v4.push_back(sc3);
v5.push_back(sc11);
v5.push_back(sc12);
v5.push_back(sc13);
v5.push_back(sc14);
foo(make_list<Base>(v1,v2)(v3)(v4)(v5));
return 0;
}
```