Creating an efficient Haskell into C++ template metaprogram compiler

Borsos Levente, Puha Márk, Szalai Norbert

2019–05–08

Abstract

There are different cases in C++ when we want to operate on data what is available at compile time. In these cases we have the option to write template metaprograms, with this we get more efficient runtime for an increased compile time or another option would be to write the corresponding functions in Haskell and call them from C++, but this has a huge overhead. For this reason, we want to develop a compiler what could generate C++ template metaprograms from Haskell so the overhead of the Haskell function calls would assimilate with the C++ compile phase achieving 100% faster runtime performance.

1 Introduction

Nowadays the use of template metaprogramming ofter occurs in modern, up-todate C++ codes. The main reason is the ability of template metaprogrmming to make complex algorithm execution at compile time. Template metaprograms are used in expression templates, static interface checking, code optimization with adaption, language embedding and active libraries. ,,However, as this capability of C++ was not a primary design goal, the language is not capable of clean expression of template metaprograms. The complicated syntax leads to the creation of code that is hard to write, understand and maintain. Despite that template metaprogramming has a strong relationship with functional programming paradigm, existing libraries do not follow these requirements. Today programmers write metaprograms for various reasons, like implementing expression templates, where we can replace runtime computations with compile-time activities to enhance runtime performance; static interface checking, which increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters, active libraries, acting dynamically during compiletime, making decisions and optimizations based on programming contexts." [1]
TODO
+ Miért lenne érdemes Haskellt használni Vagy hogyan használhatjuk a Haskellt C++ metaprogram generálásához One of the first languages which could generate C++ template metaprogram was Lambda, a Haskell-like language used to express lambda expressions [2]

```
#include <lambda.h>

__BEGIN(Haskell)
  divides b a = (a 'mod' b == 0)
  hasDivider n from to = (from <= to) && ((divides from n) ||
                                          (hasDivider n (from + 1) to));
  isPrime 1 = False;
  isPrime n = not (hasDivider n 2 (n 'div' 2));
```

```
  main = print (isPrime 1);
__END(Haskell)

#include <iostream>

int main()
{
cout << lambda::Reduce< HaskellMain >::type::value << endl;
}
```

An embedded Lambda code inside C++ program

In section 2 we discuss currently used methods to generate C++ metaprograms from Haskell(-like) code. In section 3 we present our approach of compiling Haskell code. In section 4 we compare our compiler's speed to other compilers'.

2 Current Haskell to C++ metaprogram compilers

The currently available Haskell to C++ metaprogram compilers have different methods to generate C++ metaprograms, but share strong disadvantages like functionality limitations or low speed. In this paper we discuss two popular compiling methods.

2.1 Compiling with other languages

This is the most used method to generate C++ metaprograms from Haskell code. The Haskell code goes through at least two different languages' transformation before the metaprogram code is generated from it. One functioning approach is to translate Haskell code to a similar language which will be the input for the language which will be parsed to C++ template metaprogram code. For example, C++ metaprogram code can be generated with translating Haskell code to Haskell-like *Yhc.Core* code, which is adjusted to *Lambda* language. Finally the Lambda code is used to generate C++ metaprogram code.[1] This method has high time costs because the Haskell code goes through many different transformations and translations as well as Yhc.Core's format is write only and types are not present in the language.[3]

2.2 compiling with one Haskell-like language

This method uses only one Haskell-like language to generate C++ metaprogram code. The first "Haskell-like code to C++ template metaprogram" compiler, MetaFun[4] uses this method to make C++ metaprogram code from *Kiff* language. This language substitutes Haskell to make generating template metaprograms easier and faster to code. While this method is faster than the first in terms of coding and code generating, using a Haskell-like language comes with many disadvantages: The language likely won't support all useful Haskell strategies, functions and expressions and might be not optimized well (e. g. Kiff doesn't allow currying in function calls, it has no support for lambda expressions and has zero optimization) and the new language must be learnt properly to generate fast and correct template metaprograms.

Listing 1: Definition of sum using Kiff with comments of missing functions

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f x [] = x
foldl f x (y:ys) = foldl f (f x y) ys

-- The builtin operators are not
--first-class functions
add x y = x + y
-- Currying is not yet supported
sum xs = foldl add 0 xs
```

3 Compiling from Haskell directly

Our compiler (FHC = Fast Haskell to C++) uses .... language and pure Haskell code to generate C++ metaprogram.

Megvalósítás vázlatos kidolgozása

A compilerünk által lefordított kódhoz rövid ismertető

hogy mi hogy lett fordítva, ez lehetne akár felsorolásban is

Our compiler was tested on small, popular Haskell functions[5] Compiling the factorial function using Lambda[1] language:

```
__lambda factorial =
  \n. (= n 0) 1 (* n (factorial (- n 1)));
```

$$\Downarrow$$

```
struct factorial;

struct factorial__implementation
{
  template <class n>
  struct apply
  {
    typedef
      lambda::Application<
        lambda::Application<
```

```
        lambda::Application<
          lambda::Application<
            lambda::OperatorEquals,
            n
          >,
          lambda::Constant<int, 0>
        >,
        lambda::Constant<int, 1>
      >,
      lambda::Application<
        lambda::Application<
          lambda::OperatorMultiply,
          n
        >,
        lambda::Application<
          factorial,
          lambda::Application<
            lambda::Application<
              lambda::OperatorMinus,
              n
            >,
            lambda::Constant<int, 1>
          >
        >
      >
    >
    type;
  };
};

struct factorial : factorial__implementation
{
  typedef factorial__implementation base;
};
```

Our compiler compiles the factorial function to a more readable, smaller C++ metaprogram code.

```
fact 0 = 1
fact n = n * fact (n - 1)
```

$$\Downarrow$$

```
template<int n> struct
fact {
```

```
    static const int value = n * fact<n - 1>::value;
};

template<> struct
fact<0> { // specialization for n = 0
    static const int value = 1;
};
```

TODO

```
all pred [] = True
all pred (head:tail) = (pred head) && (all pred tail)
```

$$\Downarrow$$

```
template<template<class> class predicate, class... list> struct
all;

template<template<class> class predicate> struct
all<predicate> {
    static const bool value = true;
};

template<
    template<class> class predicate,
    class head,
    class... tail> struct
all<predicate, head, tail...> {
    static const bool value =
        predicate<head>::value
        && all<predicate, tail...>::value;
};
```

TODO

```
or_combinator f1 f2 =
    \x -> (f1 x) || (f2 x)
```

$$\Downarrow$$

```
template<template<class> class f1, template<class> class f2> struct
or_combinator {
    template<class T> struct
    lambda {
```

```
        static const bool value = f1<T>::value || f2<T>::value;
    };
};
```

4 Comparison of compiling times

TODO...
To measure different compile times we have taken the average compile time of 100 separate runs for each method on Haskell template codes discussed in section 3. To calculate the compile time of calling Haskell functions from C++ we subtracted the compile time of the pure C++ code from the compile time of ...

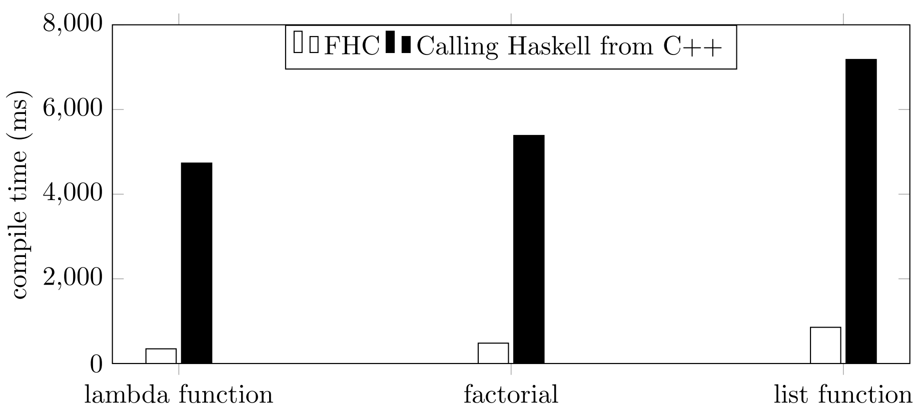

Figure 1: Compile time difference between FHC and Haskell function calling from C++

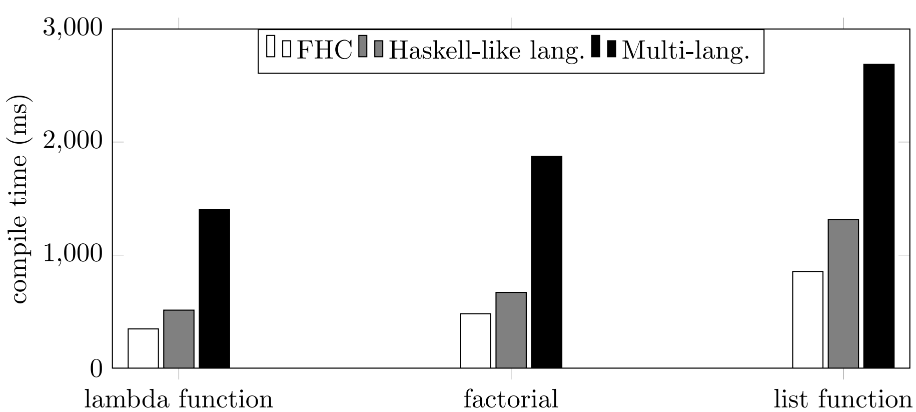

Figure 2: Compile time difference between compilers

References

[1] Z. Porkoláb and A. Sinkovics, "C++ template metaprogramming with embedded haskell," in *Proceedings of the 8th International Conference on Generative Programming & Component Engineering (GPCE 2009), ACM*, pp. 99–108, 2009.

[2] Z. Porkoláb, "Functional programming with c++ template metaprograms," in *Central European Functional Programming School*, pp. 306–353, Springer, 2009.

[3] "Yhc core - haskellwiki." `https://wiki.haskell.org/Yhc/API/Core`. Accessed: 2019-05-05.

[4] G. Érdi, "Metafun: Compile haskell-like code to c++ template metaprograms." `https://gergo.erdi.hu/projects/metafun/`. Accessed: 2019-05-05.

[5] B. Milewski, "What does haskell have to do with c++?." `https://bartoszmilewski.com/2009/10/21/what-does-haskell-have-to-do-with-c/`. Accessed: 2019-05-05.