C++ Template Metaprogramming with Embedded Haskell

Zoltán Porkoláb
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
gsd@elte.hu

Ábel Sinkovics
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
abel@elte.hu

Abstract

Template metaprogramming is an emerging new direction of generative programming: with the clever definitions of templates we can enforce the C++ compiler to execute algorithms at compilation time. Among the application areas of template metaprograms are the 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. In this paper we discuss the possibility to enhance the syntactical expressiveness of template metaprograms using an embedded functional language. Programmers can write metaprograms in Haskell syntax embedded in native C++ code and a translator tool transfers it to template metaprograms. The Haskell code snippets inter-operate with their C++ environment.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification – C++; D.3.2 [*Programming Languages*]: Language Classification – Multiparadigm languages

General Terms Languages

Keywords C++ template metaprogram, Haskell, Embedded language, Functional programming

1. Introduction

Programming is primarily is a human activity to understand the problem, make design decisions, and express our intentions to the computer. In most cases this last step manifests as writing code in a certain programming language according to its specific syntactical and semantical rules. Writing programs today is largely supported by various automated tools, like code generators mapping UML notations to source code, model driven architectures, cross-compilers, RAD tools, etc. Coding, however, is still considerably influenced by personal experiences, conventions, traditions, and customs. The syntax and the semantics of the programming language is a major factor as it seriously drives the programmer's attitude. It is possible, but not easy to program in a style which is not directly supported by the actual programming language. Even worse, if the required programming approach is not a supported paradigm. Similarly, as the spoken language has impact on human perception, the programming language may drive programmer's style. In an ideal situation the applied programming language supports the paradigm the task have to be solved in.

Templates are key language elements for the C++ programming language [3]. They are essential for capturing commonalities of abstractions without performance penalties in runtime. The most notable example is the Standard Template Library [16] is now an unavoidable part of professional C++ programs. In 1994 Erwin Unruh wrote a heavily templated program [29] in C++ which didn't compile, however, the error messages emitted by the compiler during the compilation process displayed a list of prime numbers. Unruh used C++ templates and the template instantiation rules to write a program that is "executed" as a side effect of compilation. It turned out that a cleverly designed C++ code is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [32]. These compile-time programs are called C++ *Template Metaprograms* and later has been proved to be form a Turing-complete sublanguage of C++ [9].

Today programmers write metaprograms for various reasons, like implementing *expression templates* [33], 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 [14, 19]; *active libraries* [31], acting dynamically during compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages as the AraRarat system [10] for typed safe SQL interface and `boost:xpressive` [40] for regular expressions.

Abrahams and Gurtovoy [1] defined the term template metafunction as a special template class: the arguments of the metafunction are the template parameters of the class, the value of the function is a nested type of the template called `type`. Data and even data structures can be expressed in template metaprograms with constructs like *typelist* [2].

Template metaprograms are fundamentally differ from ordinary programs. As those programs are "executed" by the compiler, we cannot speak about variables, statements, or classical control structure. In C++, in order to use a template with some specific type an instantiation is required. Hereby, the compiler creates a concrete type from a template substituting the template argument(s). This process is initiated when another code snippet refers to a template.

GPCE'09 October 4–5, 2009, Denver, Colorado, USA.
Copyright © 2009 ACM XX. . . $5.00.

As those activating code units could be templates itselves, a chain of instantiation is created under compilation. Recursive instantiations are allowed and happens in practice quite often. With the help of partial and full specialisation, we can choose between templates to instantiate, that is how Turing-complete constructions work. As objects (constant values, enums, and types) once evaluated under compilation, they will be immutable. We can still organize loops, and other complex tmplate metaprogram algorithms using recursive instantiations, and specialisations to stop recursion.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [33]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [**?**, 12].

As we can see, C++ template metaprograms' behaviour and their programming are very close to functional programming paradigm. Although, this realationship is well-known, current C++ template metaprogramming libraries does not support functional programming directly. Metaprogram implementors are forced to use alien techniques and extremely intricate syntax to implement their own concepts. This often leads to criptic, unmanagable and fragile code.

This paper is organized as follows: In section 2 we discuss the design decisions of C++ template metaprograms and their connection with functional programming. The high level overview of our transformation schema is presented in section 3. In section 4 the implementational details of the most important functional programming elements and their mapping on to C++ metaprograms is given. Tools which are essential for develop, translate and debug metaprograms written in Haskell is presented in section 6. We survey the works related in the topic 7. The paper is concluded in section 8.

2. Prime numbers

Erwin Unruh ...

Most mkd...

```
/*
 * Implementation of primes in TMPL
 */
namespace
{
  int helper_begin(char*);
}
template <int n>
struct Print
{
  enum { helper_begin_ = sizeof(helper_begin("")) };
};
template <bool condition, class True, class False>
struct If : True {};
template <class True, class False>
struct If<false, True, False> : False {};
template <bool b>
struct Bool
{
  static const bool value = b;
};
template <class a, class b>
struct And : Bool<a::value && b::value>
{};
template <int from, int to, int n>
struct IsPrimeImpl :
  If<
    from <= to,
    And< Bool<n%from != 0>,
    IsPrimeImpl<from+1, to, n> >,
    Bool< true >
  >
{};
template <int n>
struct IsPrime
{
  static const bool value =
                  IsPrimeImpl<2, n/2, n>::value;
};
struct Nop {};
template <int n>
struct PrintIfPrime :
  If< IsPrime<n>::value, Print<n>, Nop >
{};
template <class A, class B>
struct Sequence
{
  A a;
  B b;
};
template <int from, int to>
struct PrintPrimes :
  If<
    from <= to,
    Sequence<
      PrintIfPrime<from>,
      PrintPrimes<from+1, to>
    >,
    Nop
  >
{};
int main()
{
  PrintPrimes<2,20> x;
}
```

GSD: megszidom,
C runtime verzi
Haskell verzi

3. Overview of the transformation schema

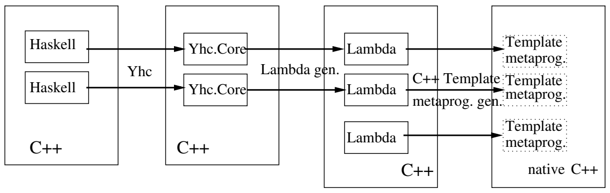

Figure 1. Overall transformation schema

3.1 Maintenance problems with template metaprograms

We have seen what template metaprogramming is capable of, but it has drawbacks as well. C++ wasn't designed to support template metaprogramming, this capability of the language was discovered later. Because of this, template metaprogramming is not a simple and easy to use tool. The syntax is intricate and error messages displayed by the C++ compilers are difficult to read and understand. Having tools supporting development of template metaprograms could let developers safely use them in production software.

We examine how functional languages could be used to write template metaprograms in, letting developers use a better syntax for writing and maintaining metaprograms. Since lambda expressions are capable of expressing any functional program we show how lambda expressions can be used to express C++ template metaprograms in. We wrote a translator which can translate nested lambda expressions into template metaprograms in C++ code.

4. Implementation

In this section we discuss the technical details of translating Haskell programs into template metaprograms. The transformation takes three steps. First Haskell code is translated to *Yhc.Core* with the Yhc compiler. Then Yhc.Core is adjusted to our *Lambda* language. And the last step, Lambda is used to generate standard compliant C++ source. Users may compile the final result with any recent C++ compiler.

4.1 Generating Yhc.Core code

Yhc.Core [15, 37] is a core Haskell-like language all Haskell programs can be expressed in. It uses a small amount of structures making it easy to process programs further. Haskell programs can be transformed into Yhc.Core using the York Haskell Compiler using the `--showcore` argument. It generates a human readable code which is easy to use for further processing. The Core language can be treated as a subset of Haskell with restrictions:

- Case statements examine their outermost constructor
- Does not contain type classes
- Does not contain `where` statements
- Has only top level functions
- Fully qualified names
- Constructors and primitives are fully applied

Currently lambda expressions are guaranteed not to appear in the output of the Haskell to Core transformation. The syntax of Yhc Core is found in [15].

The code generated by Yhc contains function definitions. Each function definition may have any number of arguments (including zero) and an `<expression>` we have defined as it's body.

4.2 Generating lambda expressions

We defined our Lambda language to express lambda expressions in a handy way. Lambda is a full-featured language. Programmers may embed Lambda code into C++ [**?**] and generate C++ template metaprograms. However, this case Lambda is used as an intermediate language.

We use the definition of non-typed enriched lambda expressions from [26]. We express the λ symbol with the \ character. As you can see our solution supports naming lambda expressions. The syntax is the following:

```
<named lambda expression> ::=
  __lambda <name> = <expression>;

<expression> ::=
  <constant> | <variable> |
  <expression> <expression> |
  \ <name> . <expression> |
  ( <expression> );
```

Decimal numbers and built-in operators are valid constants. Supported operators are: $+$, $-$, $*$, $/$, %, $<$, $>$, $<=$, $>=$, $<>$, $=$, \$. (These operators have the usual meanings, % is modulo and \$ is the fixpoint operator). We restrict the form of a general lambda abstraction allowing only one variable, i.e. the expression `\xy.E` should be written in form of `\x.\y.E`. This restriction doesn't affect expressiveness.

The code generated by Yhc contains a list of function definitions. Each function definition is converted into a named lambda expression with a corresponding name. Functions taking arguments are converted into lambda abstractions: a new abstraction is introduced for each argument of the function. These lambda abstractions wrap each other in their order appearance in the argument list. The lambda abstraction generated for the leftmost argument is the outermost. The body of the lambda abstraction generated for the rightmost argument is the lambda abstraction the body of the function definition is transformed into.

Function applications are handled by our lambda expressions. The `let` expressions and the `case` expressions are transformed into lambda expressions supported by our syntax based on the transformation techniques described in [26].

4.3 Generating template metaprograms

We have another tool transforming lambda expressions into C++ template metaprograms which can be compiled by any standard C++ compiler. These metaprograms have access to natively implemented template metaprograms making interoperatability between lambda expressions (and because of this Haskell functions) and natively implemented template metaprograms possible.

During the execution of the generated template metaprograms the C++ compiler builds the graph of the expression and reduces it lazily. Our compiler compiles named lambda expressions into C++ classes (metafunction classes [1]) implementing the lambda expression. The names of the classes are the names of the lambda expressions indicating that names have to be valid C++ names. Since these expressions are translated into C++ classes they can be at any part of the code where classes can be defined [3] indicating that Haskell code can be embedded at any part of the C++ code where classes can be defined.

Lazy and eager evaluation Our compiler supports lazy evaluation of lambda expressions: every (sub)expression is evaluated only when it's value is needed. It makes implementation of infinite data structures (such as infinite lists) possible. Eager evaluation is supported by the classes implementing the lambda expressions in C++ but are not supported directly in the lambda expressions themselves: they are always evaluated lazily indicating that Haskell functions are always evaluated lazily.

Currying Currying is supported: when the number of elements applied to a function symbol is less than the number of elements required by the function symbol the result is a new function symbol. For example: we have an anonymous function requiring two elements to be applied to it: `\x.\y. + x y`. When only one element is applied to this function the result is a new function requiring one element to be applied to it. `(\x.\y. + x y) 5` is equivalent to `\y. + 5 y`.

The C++ template metaprogram equivalent of these lambda expressions supports currying as well. Currying has to be used explicitly: only one element can be applied to the metaprogramming equivalent of a function at a time. Applying one element to the equivalent of a function requiring multiple elements being applied to it is evaluated to the equivalent of another function requiring less (by one) elements. Another element needs to be applied to that function after that, etc. The same thing happens in lambda expressions in a series of applications, for example in `f 5 8`.

Haskell function applications are translated into applications of lambda expressions indicating that the template metaprograms genereated from Haskell functions support currying and are evaluated using currying.

Recursion Named lambda expressions translated to template metafunctions can reference themselves:

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

Our compiler generates the following code from this example:

```
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;
};
```

Constants Constants are implemented by a class. Currently two types of constants are supported: integral constants and types. Types are implemented by themselves, for example the type `int` is implemented by `int`. Integral constants are implemented by a wrapper class, such as the wrappers from `boost::mpl` [1]. Currently Haskell code can't reference types, it has access to integral constants only.
Lambda abstractions Lambda abstractions are implemented by metafunction classes [1] whose embedded `apply` metafunction takes exactly one argument. The name of the argument is the name of the variable the lambda abstraction bounds.

For example here is a lambda expression and it's implementation:

```
// The lambda expression
__lambda I = \x. y;

// It's implementation
struct I {
  template <class x>
  struct apply {
    typedef y type;
  };
};
```

Variables Variables are implemented by their name. A name symbol from the lambda expression becomes a name symbol in C++. Binding of the names in lambda abstractions is done by the C++ compiler. As we could see it in the previous example the lambda expression `y` becomes `typedef y type` in the C++ template metaprogram. The example has a lambda abstraction binding `x`. This lambda abstraction is represented by a template metafunction taking one argument called `x`. When this metafunction is instantiated the `x` symbols in it's body (if there are any) are replaced by the class the metafunction is instantiated with.
Eagerly evaluated applications Eager application of a lambda expression to a lambda abstraction is implemented by the evaluation of the `apply` metafunction. The C++ compiler does the β conversion during the instantiation because the name of the bounded variable is the name of the argument of the nested `apply` metafunction (and the variables are implemented by their names).

The `I` lambda expression defined in the previous section can be evaluated either in an eager or lazy way. To specify eager evaluation, the user should use the following C++ construct:

```
typedef I::apply<I>::type ApplicationOfIToItself;
```

We will discuss lazy evaluation in subsection 4.4.
Currying in built-in functions Built-in in functions (such as the arithmetical or logical operators) have more than one arguments. Their implementation has to support currying. They have to be implemented as a lambda abstraction. For example applying an element on the plus operator has to evaluate to another lambda abstraction, applying another element on that has to evaluate to a constant (and the value of it has to be the sum of the arguments). It can be implemented easily using nested types and templates. As an example here is the implementation of the plus operator:

```
struct OperatorPlus {
  template <class a>
  struct apply {
    struct type {
      template <class b>
      struct apply {
        // ... native implementation of addition,
        // possibly by boost::mpl
      };
    };
  };
}
```

We assume that every built-in function supports partial evaluation (to a lambda abstraction).

4.4 Lazy application

Applications in lambda expressions (and in Haskell) are evaluated only when their value is needed, they can't be translated into eager applications. We use the following template to implement lazy application:

```
template <class left, class right>
struct Application {};
```

Using this template expressions for lazy evaluation can be built as binary trees of applications: the instances of the `Application` template represent the application nodes of the tree, the `left` and `right` arguments represent the sub trees of the application nodes.

We define a metafunction implementing reduction of expressions to weak head normal form [5]. Stand alone lambda abstractions, constants and built-in functions are in weak head normal form. Lazy applications are never in weak head normal form, since we assume that every built-in function supports partial evaluation. These considerations simplify the reduction algorithm:

```
while (the top level element is a lazy application)
  reduce the left side of the top level element to
    weak head normal form
  evaluate the top level application
```

We implemented this in a metafunction called `Reduce`:

```
template <class T> struct Reduce {typedef T type;};

template <class left, class right>
struct Reduce< Application<left, right> > {
  typedef
    typename Reduce<
      typename
      Reduce<left>::type::template
      apply<right>::type
    >::type  type;
};
```

The general case handles lambda expressions which are already in weak head normal form, there is a specialisation of the template for reducing lazy applications in normal order reduction: it reduces the left sub-expression of the application to weak head normal form (`typename Reduce<left>::type`) after which the left side is in weak head normal form, so the next redex is this application:

```
  typename
    Reduce<left>::type::template apply<right>::type
```

Finally the resulting expression is reduced as well.

In the following example we present the whole generation process. This simple Haskell code

```
  f x = 2 * x
```

is transformed into the following Yhc.Core code:

```
  Main;f v219 v220 =
    let ATOM221 = (Prelude;fromInteger v219) 2
    in ((Prelude;*) v219) ATOM221 v220
```

Our tool transforms it to

```
  Main_f = \v219. \v220.
    (\ATOM221. * v219 ATOM221 v220)
    (Prelude_fromInteger v219 2)
```

4.5 Interoperability with native C++ metafunctions

Lambda expressions have C++ equivalents and they can be implemented natively as well. Natively implemented lambda expressions can be used in lambda expressions (as constants). For example:

```
  struct NativeLambdaExpression {
    // native implementation...
  };

  __lambda f = \n. NativeLambdaExpression 2 n;
```

It makes extension of the built-in operators possible and parts of the expressions can be implemented using other techniques.

Lambda expressions can be used by native C++ template metaprograms as well since lambda expressions are compiled into template metaprograms. After they are compiled into template metaprograms there is no difference between a natively implemented lambda expression and a compiled one: the compiled one can be used as a natively implemented one. Lambda expressions can be used as built-in functions in other lambda expressions, for example:

```
  __lambda add = \a.\b. + a b;
  __lambda f = \n. * n (add 6 7);
```

Lambda expressions can be used in their own definition simplifying the creation of recursive expressions:

```
  __lambda rec = \n. (< n 1) 13 (rec (- n 1));
```

Due to the visibility rules of C++ [3] lambda expressions are visible after their declaration. For example the following code wouldn't compile because `b` is defined after `a`:

```
  __lambda a = \n. b n;
  __lambda b = \n. + 1 n;
```

Our compiler supports forward declaration of lambda expressions by ensuring that every lambda expression compiled to C++ will be implemeneted as a `struct`. The previous example `b` can be declared before `a`:

```
  struct b;
  __lambda a = \n. b n;
  __lambda b = \n. + 1 n;
```

Haskell functions are visible in the whole compilation unit, to support this our Yhc.Core to lambda expression transformation tool adds forward declaration of the named lambda expressions to the beginning of each lambda expression list generated from an embedded Haskell block. Note that this makes function visible to each other within an embedded Haskell block. Visibility of functions defined in separate Haskell blocks depend on the C++ visibility rules [3] because Haskell functions are transformed into C++ classes.

5. Evaluation

We solved the same problem with a hand-written C++ Template Metaprogram and with embedded Haskell. The task was determining about an natural number wether it's a prim or not. We used a simple linear algorithm: for a natural number `n` we test whether numbers `[2..n/2]` are dividers of `n` or not. During the tests we run the algorithm for primes only to ensure that all numbers in the `[2..n/2]` range are tested. Using greater primes the number of tests done by the algorithm grows letting us see how each of the tools scale.

5.1 Code size

We have compared the length of the code to write (debug and maintain) by counting the effective lines of code. Our native implementation was 42 lines long while the solution using embedded Haskell was only 9 lines long. It means that using embedded Haskell reduces the length of the code - in our experiment the difference was significant.

5.2 Template depth

We have compared how deep template depth the two solutions require. The embedded Haskell solution exceeded the (default)

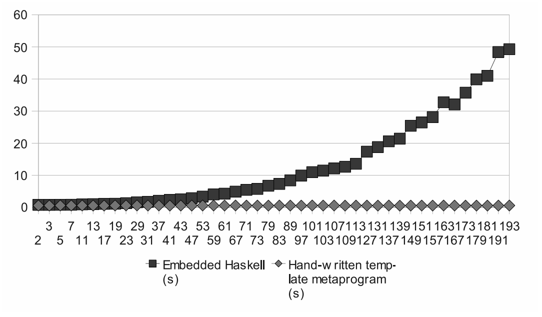

Figure 2. Compilation time

maximum template depth of the GNU C++ compiler for primes greater than 193 while the native implementation exceeded the maximum only for primes greater than 331. As we can see the limit of the embedded Haskell solution is lower (because embedded Haskell is based on currying).

5.3 Compilation time

We have measured compilation time of generated metaprograms and compared it with a native implementation (using the same algorithm). We run the tests on a Linux PC with 1 GB memory and a 2.6 GHz Celeron CPU. We used the 4.2.4 version of the GNU C++ compiler with default options. We used the `time` command to measure compilation time and used the `user` part of it's output.

Figure 2 shows the compilation times (the horizontal axis is the length of the interval (the first element of the interval was always 2), the vertical axis shows the seconds spent on compilation.

6. Essential tools

Maintaining multiparadigm, multilanguage solutions face a number of difficulties. Designers and implementors have to think according to more than one programming paradigms. Developers have to be familiar with multiple languages. Defining a good interface between languages and paradigms is not trivial either.

Debugging mixed language programs is one of the major challenges. Our solution doesn't make debugging metaprograms easier, the error messages are about the C++ templates representing the lambda expressions. But having a translation of lambda expressions to template metaprograms gives opportunities of extending the translated code with information about the original lambda expression making error messages more descriptive for developers.

Let suppose the following Lambda code, which has a hard to detect bug.

```
#include <lambda.h>
#include <iostream>

__lambda fib =
  \n.
    (== n 0)
    1
    (
      (== n 1)
        1
        (+ (fib (- n 1)) (fib (- n 2)))
    )
;

__lambda fib5 = fib 5;

int main()
{
  std::cout << lambda::Reduce<fib5>::type::value
             << std::endl;
}
```

The root of the problem is that the equal operator = has been written as ==. As our Lambda language is not typed, the problem has not been detected until the generated C++ metaprogram is executed, i.e. the C++ compiler starts to instantiate them.

The compilation process will produce the following diagnostics:

```
./lambda.h: In instantiation of 'lambda::Operator
Equals::apply<lambda::OperatorEquals>::type::apply
<lambda::Constant<int, 5> >':
./lambda.h:174:   instantiated from 'lambda::Reduce
<lambda::Application<lambda::Application<lambda::
OperatorEquals, lambda::OperatorEquals>, lambda::
Constant<int, 5> > >'
./lambda.h:174:   instantiated from 'lambda::Reduce
<lambda::Application<lambda::Application<lambda::
Application<lambda::OperatorEquals, lambda::
OperatorEquals>, lambda::Constant<int, 5> >, lambda
::Constant<int, 1> > >'

//.. other 50 lines ...

./lambda.h:174:   instantiated from 'lambda::Reduce
<lambda::Application<fib, lambda::Constant<int, 5>
> >'
./lambda.h:146:   instantiated from 'lambda::
ReduceBase<fib5>'
./lambda.h:45:   instantiated from 'lambda::If<true,
lambda::ReduceBase<fib5>, lambda::Identity<fib5> >'
./lambda.h:159:   instantiated from 'lambda::
Reduce<fib5>'
tmp.cpp:29:   instantiated from here
./lambda.h:280: error: 'value' is not a member of
'lambda::OperatorEquals'
tmp.cpp: In function 'int main()':
tmp.cpp:29: error: 'struct lambda::Reduce<fib5>::
type' is not a class or namespace
```

These errors are not really human readable. However, every lines in the messages above represent one reduction step of the Lambda language. An automatic process is able to parse these messages and able to reproduce the Lambsa source. For example,

```
  struct lambda::Reduce<fib5>::type
```

in the last line identifies `fib5` Lambda symbolum. The previous line

```
  lambda::Reduce<lambda::Application<
                fib,lambda::Constant<int,5> > >
```

identifies Lambda expression `fib 5`. From the error messages above our debugger is able to reproduce the original Lambda code.

```
= = 5
= = 5 1
= = 5 1 1
= = 5 1 1 (+ (fib (- 5 1)) (fib (- 5 2)))
= 5 0 1 (= = 5 1 1 (+ (fib (- 5 1)) (fib (- 5 2))))
fib 5
fib5
```

The restriction of this solution is that it shows the reduction steps of the only problematic part of the core Haskell code. Those reductions, which does not generate error messages.

To emit diagnostics for all the instantiation steps we can use external tools, like *templight*, a C++ template metaprogram debugger [18]. Templight instruments C++ code – here the final one, generated from code with embedded Haskell – and injects small templated snippets, which emit warnings on instantiations. These warnings can be collected, analyzed and used to reproduce the original instantiation chain.

Merging the facilities of templight and embedded Haskell translation is one of the most important future work.

7. Related work

7.1 FC++

FC++ is a C++ library providing runtime support for functional programming [25]. Using the tools the library provides functional programs can be written in C++ from which the expression graph is built and evaluated at runtime. They don't require any external tool (such as a translator) they use standard language features only. The library focuses on runtime execution.

7.2 Boost metaprogramming library

Boost has a template metaprogramming library called `boost::mpl` which implements several data types and algorithms following the logic of STL [12]. Our solution is designed to be compatible with it (the lambda expressions produced by our compiler are designed to be template metafunction classes taking one argument).

`Boost::mpl` has lambda expression support: the library provides tools to create lambda abstractions easily: placeholders (`_1`, `_2`, etc.) are provided and arguments of metafunctions can be replaced by them. The result of evaluating a metafunction with one (or more) placeholder argument is not directly usable, a metafunction called `lambda` generates a metafunction class from them. Using these lambda abstractions partial function applications can be implemented, but since `lambda` bounds every placeholder lambda abstractions with other lambda abstractions as their value can't be defined. For example λx.λy.+xy can't be expressed (and neither can be the `Y` fixpoint operator).

7.3 Boost lambda library

Boost has a library for implementing lambda abstractions in C++ [39]. It's main motivation is simplifying the creation of function objects for generic algorithms (such as STL algorithms). With the library function objects can be built from expressions (using placeholders). The lambda abstractions built using this library can be used at runtime.

7.4 Haskell type classes

Zalewski et al. defined a mapping from generic Haskell specifications to C++ with concepts [35]. Haskell multi-parameter type classes with functional dependencies have been translated to ConceptC++, an experimental implementation of the concept feauture of C++0x. The translation process consists of three major parts: the division of Haskell class variables i nto ConceptC++ concept parameters and associated types, the corresponding division of superclasses in the context of a type class, and the flattening of Haskell AST to the concrete syntax of ConceptC++. The main motivation of the authors was to model software components in Haskell and implemented in C++ automated the translation.

7.5 Projects based on Yhc.Core

There are a number of other projects are based on Yhc.Core. Most notably, YCR2JS is a Converter of Yhc Core to Javascript [38] which generates Java Script from Haskell, similarly as we generate C++ code.

7.6 EClean

Our solution is not the first attempt to express template metaprograms using a functional language. A Clean to Template Metaprogram translator has been written [28]. It uses a subset of Clean (EClean) as the source language which it creates template metaprograms from.

8. Conclusion

Ideally, the syntax of a programming language should match to the paradigm the program is written in. Template metaprogramming, a Turing-complete subset of the C++ language for implementing compile-time algorithms via cleverly placed templates, is many times regarded as a pure functional language. Unfortunately, the current way of writing metaprograms is far from the ideal, mainly due to the complicated template syntax and the different original design goals of C++.

In this paper we introduced a method which makes metaprogram developers able to express their intentions directly in functional style using Haskell syntax. Haskell code snippets are embedded into the C++ program and are translated into native C++ code. The translation process uses a stepwise approach; first compile Haskell to *Yhc.Core* using Yhc compiler, then our tool generates lambda expressions as an intermediate representation of core Haskell. In the last step lambda expressions are used to generate C++ template metaprograms. This way we get a homogeneus C++ program which could be compiled by any standard C++ compiler.

As maintaining multiparadigm and embedded languages always a challenge, we provided some essential tools to debug the generated C++ metaprograms help controlling the transformation process. Especially, we provided a "debugger" which displays the reduction steps of errorneous core Haskell.

We have shown that using embedded Haskell simplifies template metaprograms, make them easier to write and maintain. The developer can focus on the functionality of the metaprogram, reusing a huge number of existing algorithms and data structures make them available to the C++ metaprogramming community.

References

[1] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2004.

[2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[3] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[4] T. H. Brus, C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer, *CLEAN: A language for functional graph rewriting*, Proc. of a conference on Functional programming languages and computer architecture, Springer-Verlag, 1987, pp.364-384.

[5] Zoltán Csörnyei and Gergely Dévai, *An introduction to the lambda-calculus*, Lecture Notes in Computer Science, Springer-Verlag, LNCS Vol. 5161, pp. 87-111 ISSN 0302-9743, ISBN 3-540-88058-5

[6] Zoltán Csörnyei, *Lambda kalkulus – A funkcionális programozá alapjai* Typotex, 2007, Budapest, ISBN: 978-963-9664-46-3

[7] Olaf Chitil, Zoltán Horváth, Viktória Zsók (Eds.): *Implementation and Application of Functional Languages*, Springer, 2008, [273], ISBN: 978-3-540-85372-5

[8] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. L. Veldhuizen, *Generative Programming and Active Libraries*, Springer-Verlag, 2000.

[9] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[10] Yossi Gil, Keren Lenz, *Simple and Safe SQL queries with C++ templates* In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.

[11] Zoltán Horváth, Rinus Plasmeijer, Anna Soós, Viktória Zsók (Eds.): *Central European Functional Programming School*, Springer, 2008, [301], ISBN: 978-3-540-88058-5

[12] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.

[13] P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers, *Functional programming in Clean*, 2002

[14] Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000

[15] N. Mitchell, C. Runciman, *A Supercompiler for Core Haskell*, In Chitil et al. Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers, Springer-Verlag, Berlin, Heidelberg, 2008

[16] D. R. Musser, A. A. Stepanov, *Algorithm-oriented Generic Libraries*, Software-practice and experience 27(7), 1994, pp.623-642.

[17] R. Plasmeijer, M. van Eeekelen, *Clean Language Report*, 2001.

[18] Zoltán Porkoláb, József Mihalicza, Ádám Sipos, *Debugging C++ template metaprograms*, In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM 2006, ISBN 1-59593-237-2, pp. 255-264.

[19] Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First Workshop on C++ Template Metaprogramming, October 2000

[20] J. Siek, A. Lumsdaine, *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.

[21] Douglas Gregor, Jaakko Jrvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. In Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.

[22] J. Siek, *A Language for Generic Programming*, PhD thesis, Indiana University, 2005.

[23] B. Stroustrup, *The C++ Programming Language Special Edition*, Addison-Wesley, 2000.

[24] G. Dos Reis, B. Stroustrup, *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.

[25] B. McNamara, Y. Smaragdakis, *Functional programming in C++*, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.

[26] Simon L. Peyton Jones: *The Implementation of Functional Languages*, Prentice Hall, 1987, [445], ISBN: 0-13-453333-9 Pbk

[27] Ábel Sinkovics, Zoltán Porkoláb: *Expressing C++ Template Metaprograms as Lambda expressions*, In Tenth symposium on Trends in Functional Programming (TFP '09, Zoltn Horvth, Viktria Zsk, Peter Achten, Pieter Koopman, eds.), Jun 2 - 4, Komarno, Slovakia 2009., pp. 97-111

[28] Ádám Sipos, Zoltán Porkoláb, Viktória Zsók: *Meta<fun> – Towards a functional-style interface for C++ template metaprograms* In Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII, 2008/2, Cluj-Napoca, 2008, pp. 55-66.

[29] E. Unruh, *Prime number computation*, ANSI X3J16-94-0075/ISO WG21-462.

[30] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.

[31] Todd L. Veldhuizen and Dennis Gannon: Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23

[32] T. Veldhuizen, *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[33] T. Veldhuizen, *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[34] T. Veldhuizen, *C++ Templates are Turing Complete*

[35] M. Zalewski, A. P. Priesnitz, C. Ionescu, N. Botta, and S. Schupp, *Multi-language library development: From Haskell type classes to C++ concepts*. In MPOOL 2007 Ecoop workshp, 2007.

[36] I. Zólyomi, Z. Porkoláb, *Towards a template introspection library*, LNCS Vol.3286 (2004), pp.266-282.

[37] The Yhc wiki,
http://www.haskell.org/haskellwiki/Yhc

[38] The Yhcr2js homepage,
http://www.haskell.org/haskellwiki/Yhc/Javascript

[39] Boost Libraries.
`http://www.boost.org/`

[40] The boost xpressive regular library.
`http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html`