Point-Free Templates

European LLVM Developers Meeting 2018

Andrew Gozillon, Dr. Paul Keir

School of Engineering and Computing
University of the West of Scotland, Paisley Campus

April 17th, 2018

Introduction

- C++ template metaprogramming is similar to programming in functional languages.
- This leads to the question: what functional features can be leveraged to make template metaprogramming more powerful? We believe that currying is one of these features.
- Currying can be used to make point-free functions, which we believe can be used as type-level lambda functions.
- Using Clang's LibTooling we created a tool for translating pointful templates into point-free templates.

Currying

- Currying is a feature common to functional languages, for example: Haskell, F# and OCaml.
- Currying treats a multi-parameter function as a sequence of one argument functions.
- If not enough arguments are applied to a function to meet the criteria for evaluation it results in a new function requiring less arguments rather than a failed application.
- This is called partial application.

Currying Example

The specialization of a function with a fixed argument:

```
add :: Num a => a -> a -> a
add x y = x + y

addTwo :: Num a => a -> a
addTwo = add 2

addTwo 3 => 5
```

Code Simplification, Uncurried vs Curried:

```
map (\ x -> add 2 x)  [1, 2, 3]
map (add 2) [1, 2, 3]
```

Point-Free

- Point-free programming is a style that removes parameters from a functon.
- It does this through the use of combinators and higher-order functions.
  - Higher-order functions: Can take functions as arguments or return functions as a result.
  - Combinators: A type of higher-order function that can be used to manipulate the logic of functions and can help combine them in unique ways.
- Overall this can make functions more concise and equational.
- Currying helps enable programming techniques like the point-free style.

Point-Free Example

```
returnRight x y = y
returnRight = const id

f x y = returnRight y x
f = flip returnRight

f = flip (const id)
```

Goal

- Lambdas are a useful language feature removing the requirement to create named implementations of functions, making code more concise.
- This is quite handy with higher-order functions. For example in Haskell:

```
map (\x -> x + 1) [1,2,3] => [2,3,4]
```

- Similar example with values in C++:

```
std::transform(foo.begin(), foo.end(), bar.begin(),
      std::function<int(int)>([](int x) { return x + 1; }));
```

- Currently C++ doesn't have Type-Level lambdas, so you can't do the same when metaprogramming.
- Could we provide something similar using point-free templates?

Currying in our Template API

- To curry templates we make use of our C++ Template API.
- The main user level API functions are:
  - *quote_c* - stores a meta-function for invocation and directly invokes the "type" member of a meta-function when evaluated.
  - *quote* - stores a meta-function for invocation, it will not invoke the "type" member of the meta-function. An intermediate type alias should instead be supplied in place of the meta-function to access specific members.
  - *eval* - evaluates a passed in meta-function and a set of arguments. The result can be the final evaluated result of the meta-function or a curried template if more arguments are required.

Currying with our Template API

```
template <typename T1, typename T2>
struct common {
   using type = typename std::common_type<T1, T2>::type;
};

template <typename T1, typename T2>
using common_t = typename common<T1, T2>::type;

using common_q = quote<common_t>;
using common_c = quote_c<common>;
using curried = eval<common_q,int>;

static_assert(std::is_same_v<float,eval<common_q,int,float>>);
static_assert(std::is_same_v<float,eval<common_c,int,float>>);
static_assert(std::is_same_v<float,eval<curried,float>>);
```

The Template APIs Combinators and Higher-Order Functions

- We make use of several different combinators from our template API to support point-free templates.
- Many of these are implemented in Haskell and other functional languages.
- The main combinators we make use of to support point-free templates are:
  - $id(x) = x$
  - $const(x, y) = x$
  - $flip(x, y, z) = x(z, y)$
  - $compose(x, y, z) = x(y(z))$
  - $S(x, y, z) = xz(yz)$
- The S, $const$(K) and id(I) combinators are known as the SKI combinator calculus and they can function as a simple turing complete language.

A Point-Free Template Example

```
// Previous example made point-free
using commonPf = quote_c<std::common_type>;

template <typename T, typename T2>
struct returnT2 {
    using type = T2;
};

using returnT2Pf = eval<const_,id>;

template <typename F, typename X, typename Y, typename Z>
struct flip3 {
  using type = invoke<F,X,Z,Y>;
};

using flip3Pf = eval<compose,compose,flip,quote<invoke>>;
```

Point-Free Libtool

- You can see that point-free functions can get quite complex.
- Weve developed a Clang Libtool to translate templates into point-free form.
- It's based on and borrows from existing open source Haskell point-free translators.

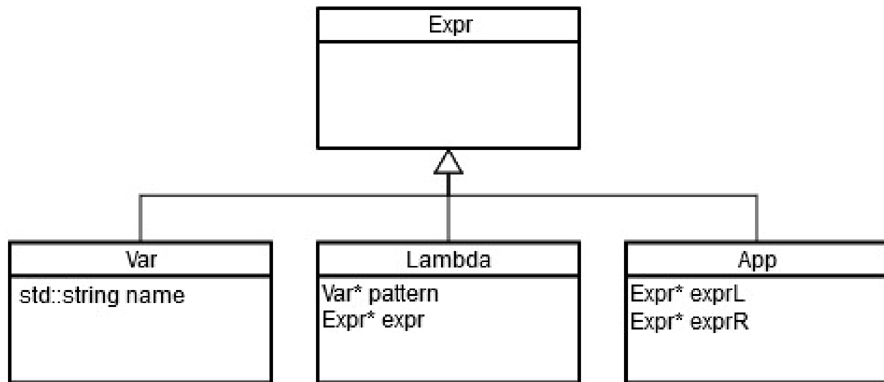
- Used from the command line it takes as input:
  - A C++ Source or Header File.
  - The name of the template class the user wishes to target.
  - The name of the member type alias or type definition the user wishes converted (will default to "type", if nothing is supplied).
- In return it will print to the command line the point-free variation of the template.

Point-Free Pipeline

Intermediate Lambda Calculus

- This tool makes use of a simple untyped (even though we work with types) lambda calculus.
- Which contains three simple constructs:
  - Variable - represent ordinary variables (types in this case) or operators like multiplication.
  - Lambda Abstraction - anonymous functions which take a variable and contain an expression.
  - Application - represent the application of a lambda abstraction to a result or variable, like passing an argument to a function.

Intermediate Lambda Calculus



- Stringified Example: *(Lambda (PVar T) (App (Lambda (PVar T2) (Var T2)) (Var T)))*

Template to Lambda Example

```
template <typename T>
struct Foo {
    using type = int;
};

template <typename T>
struct Bar {
     using type = typename Foo<T>::type;
};
```

- Foo: *(Lambda (PVar T) (Var int))*
- Bar: *(Lambda (PVar T) (App (Lambda (PVar T) (Var int)) (Var T)))*
- Type Traits are handled a little differently than user defined templates like Foo.
  - Instead of breaking them down into a lambda they're treated like another variable that can have variables applied to them.
  - This is because some type traits have built in compiler support like *is_polymorphic*.

Template to Lambda Implementation

- The primary use of Clang/LLVM within this libtool is to parse the AST looking for information of interest to populate our lambda calculus with.
- Most of the heavy lifting is done by a recursive function named TransformToCExpr which progresses down the AST from a passed in Node generating a returnable lambda calculus in the process.
- There are four overloaded versions of TransformToCExpr for Clang Type's, Expr's, Decl's and NestedNameSpecifier's.

Template to Lambda Implementation

Program Steps:

1. Push initial template name and member name onto stack.
2. Retrieve and search the TranslationUnitDecl recursively for the ClassTemplateDecl referred to by the name on the stack.
3. When found invoke TransformToCExpr on the ClassTemplateDecl.
4. TransformToCExpr traverses the ClassTemplateDecl looking for important information for generating the lambda calculus.
5. We also look for the next Node we are interested in referred to by the member name and pass that to TransformToCExpr.
6. We continue in this manner recursively going deeper down the AST rooted at the original ClassTemplateDecl building our Lambda representation.

Template to Lambda Implementation

The main Clang AST nodes of interest:

- ClassTemplateDecl - Generates one or more nested lambdas depending on the number of template parameters, the type nested inside makes up the last lambdas Expr e.g. *(Lambda (PVar T) (Lambda (PVar T) (Var int)))*.
- TemplateSpecializationType - Loops over all of the arguments given and invokes a transform on each. This generates an App if multiple arguments are given or a Var if a single argument is given e.g. *(App (Var Foo) (Var T))*.
- PointerType - Creates an App with a pointer on the right side and the type its been applied to on the left side e.g. *(App (Var int) (Var *))*.
- BuiltinType - Creates and returns a Var containing the name of a hard-coded type like int or float e.g. *(Var int)*.
- PackExpansionType - Creates and returns a Var that notates it's a pack expansion e.g. *(Var ...T)*.
- TemplateTypeParmType - Creates a Var with the name of the template type parameter e.g. *(Var T)*.

Point-Free Algorithm

- The goal of point-free algorithms is to detect patterns and remove variables within code (or a calculus) by rearranging and applying a series of curried combinators and higher-order functions so that the result has the same semantics but is point-free.
- Several tools exist in functional languages to achieve this goal, for example Haskell has the webtool Pointfree.io and the point-free tool from Haskell's hackage.
- Our point-free algorithm is based on hackage's point-free tool.

Point-Free Algorithm

The main section of the algorithm deals with removing lambda parameters from expressions and adding certain curried functions based on node entered and patterns found:

- We seek to remove Lambdas by traversing their expression and removing the parameter where it's found.
- For Applications we check to see which side the parameter is contained and then traverse the expression before applying a combinator:
  - Left: Wrap an Application of *Flip* around the left and right expressions.
  - Right: Wrap an Application of *Compose* around the left and right expressions.
  - Left & Right: Wrap an Application of *S* around the left and right expressions.
  - Not found: Wrap an Application of *Const* around the existing Application.
  - Special Case: If the right expression is a variable and is the parameter we can remove it.
- We replace Variables with *Id* if we find that it contains the parameter we wish to modify and *Const* otherwise.

Lambda to Point-Free Template

- After the Lambda Calculus has been made point-free the lambda to template conversion is quite simple.
- We just go through the point-free lambda and build a string using the Template API functionality where it's called for.

For example:

- Point-Free Lambda: *(App (Var flip) (App (Var const) (Var id)))*
- Point-Free Template: *eval<flip, eval<const_, id>>*

Conclusion

We have:

- Applied existing algorithms and concepts in the functional programming world to templates.
- This has allowed for point-free template code through our template API that enables currying.
- Created a tool for translating pointful templates to point-free templates; that can be used in lieu of type level lambdas.

Future work:

- Point-Free template specializations.
- Support for non-type template parameters.