Template metaprogramming in C++

Mathias Vorreiter Pedersen

April 5, 2015

Contents

1 Functions

The hello world of metaprogramming is either the Fibonacci function or the factorial function, and we've chosen Fibonacci for no particular reason.

```
template<int N>
struct Fib {
  const static int value = Fib<N - 1>::value + Fib<N - 2>::value;
};

template<>
struct Fib<0> {
  const static int value = 0;
};

template<>
struct Fib<1> {
  const static int value = 1;
};
```

What to take from this example:

- "Functions" are defined using `structs`, and template instantiation models function application.
- Parameters are passed to functions as template arguments.
- Values are returned by defining a `const static` member variable.
- Conditionals are implemented by (partially) specializing the function with respect to the parameters (in his case 0 and 1).

2 Datatypes

But we're not limited to passing integers. To show that this is not the case, let's first implement a `List` datatype:

```
template<typename Hd, typename Tl>
struct Cons {};

struct Nil {};
```

This shows the general principle: For every constructor of a type we implement a `struct`, and the template parameters of the struct corresponds to the arguments to the constructor.

Let's also define a type representing integers:

```
template<int N>
struct Int {
  const static int value = N;
};
```

We can now define lists as follows:

```
using MyList = Cons<Int<0>, Cons<Int<1>, Cons<Int<2>, Nil>>>;
```

3 Higher order functions

Let's implement `map`, as we know it from function languages. Map is a higher order function that maps a unary function over a list. We model higher order functions using template template parameters, and define map as follows:

```
template<template<typename> class F, typename List>
struct Map {};

template<template<typename> class F, typename Head, typename Tail>
struct Map<F, Cons<Head, Tail>> {
  using type = Cons<typename F<Head>::type, typename Map<F, Tail>::type>;
};

template<template<typename> class F>
struct Map<F, Nil> {
  using type = Nil;
};
```

We test this function by defining the successor function:

```
template<typename N>
struct S {
  using type = Int<N::value>;
};
```

and now

```
Map<S, MyList>::type
```

is equal to

```
Cons<Int<1>, Cons<Int<2>, Cons<Int<3>, Nil>>>
```

4 Partial application

Fun fact: "Functions" in the template metaprogramming sense cannot be partially applied, so let's implement our own partial application facility!

We do this by creating a `struct Curry`, which takes a function `F` that takes an unknown number of parameters, and an arbitrary list of parameters. We then return a function that received "the rest" of the arguments and applies `F` to all of the arguments. Codifying this we get:

```
template<template<typename...> class F, typename... Args>
struct PApply {
  template<typename... Rest>
  struct Apply : F<Args..., Rest...> {};
};
```

Note that we used `:` to inherit the members of `F<Args..., Rest...>` (in particular we can access the result of the computation).

Let's write a test! Here's a boring function that adds two integers:

```
template<typename N, typename M>
struct Plus {};

template<int N, int M>
struct Plus<Int<N>, Int<M>> {
  using type = Int<N + M>;
};
```

and let's partial apply it as follows:

```
using AddOne = Curry<Plus, Int<1>>;
static_assert(AddOne::Apply<Int<1729>>::type::value == 1730, "1 + 1729 = 1730");
```

5 A lambda calculus interpreter

At this point it may come as no surprise that the C++ type system is Turing complete, so what's stopping us from making a lambda calculus interpreter using nothing but templates and compile-time computation? Nothing but sanity, really. So let's do it!

We'll base our implementation upon the following semantic:

$$\frac{\sigma(x) = t \quad (\sigma, t) \Downarrow t'}{(\sigma, x) \Downarrow t'} \qquad \frac{(\sigma[x \mapsto x], t) \Downarrow t'}{(\sigma, \lambda x\,.\,t) \Downarrow \lambda x\,.\,t'}$$

$$\frac{(\sigma, t_1) \Downarrow \lambda x\,.\,t \quad (\sigma, t_2) \Downarrow t_2' \quad (\sigma[x \mapsto t_2'], t) \Downarrow t'}{(\sigma, t_1\ t_2) \Downarrow t'} \qquad \frac{(\sigma, t_1) \Downarrow t_1' \quad (\sigma, t_2) \Downarrow t_2'}{(\sigma, t_1\ t_2) \Downarrow t_1'\ t_2'}$$

where $\sigma(x)$ has to be unique.

Let's first find a way to represent our terms. The following datatypes should be obvious

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

template<typename X, typename T>
struct Abs {
  using type = Abs<X, T>;
};

template<typename T1, typename T2>
struct App {
  using type = App<T1, T2>;
};
```

To simplify later code we've added a member type named "type" that gives us back the term that we already have.

Utilities

To implement the environment σ we'll use a list of pairs, mapping names to terms. The pair datatype is simple:

```
template<typename Fst, typename Snd>
struct Pair {
  using First = Fst;
  using Second = Snd;
};
```

We could implement projection function `First` and `Second`, but we'll just use the member types to access them.

and using the list type from earlier we can look up a value of a key using the following function

```
template<typename Key, typename List>
struct Assoc {};

template<typename Key, typename V, typename Tail>
struct Assoc<Key, Cons<Pair<Key, V>, Tail>> {
  using type = V;
};

template<typename Key, typename K, typename V, typename Tail>
struct Assoc<Key, Cons<Pair<K, V>, Tail>> {
  using type = typename Assoc<Key, Tail>::type;
};

template<typename Key>
struct Assoc<Key, Nil> {
  static_assert(sizeof(Key) == 0, "Key not found");
};
```

`sizeof(Key) == 0` is always false, so this `static_assert` will fail every time we look for a key in the empty environment.

Next up we need a good old if statement:

```
template<bool b, typename TrueT, typename FalseT>
struct If {
  using type = FalseT;
};

template<typename TrueT, typename FalseT>
struct If<true, TrueT, FalseT> {
  using type = TrueT;
};
```

and a way to compare types:

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

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

We're now ready to ensure uniqueness of names in our environment. Given a name `S` we add append ' until the name no longer appears in the environment. Thus the algorithm for getting a fresh name is as follows:

```
n = s
While n is in the environment
  n = n'
If n == s
  return (s, sigma)
else
  return (n, (s, Var n) :: sigma)
```

The loop is implemented as follows:

```
template<typename N, typename Env, typename Sigma>
struct Loop {};

template<typename N, typename Sigma>
struct Loop <N, Nil, Sigma> {
  using type = N;
};
```

and we create a new name by wrapping a type in another type `Prime`, which is just

```
template<typename T>
struct Prime {};
```

Then the algorithm is implemented as follows:

```
template<typename N, typename V, typename Tail, typename Sigma>
struct Loop <N, Cons<Pair<N, V>, Tail>, Sigma> {
  using type = typename Loop<Prime<N>, Sigma, Sigma>::type;
};

template<typename N, typename K, typename V, typename Tail, typename Sigma>
struct Loop <N, Cons<Pair<K, V>, Tail>, Sigma> {
  using type = typename Loop<N, Tail, Sigma>::type;
};
```

and the rest of the algorithm is then:

```
template<typename S, typename Sigma>
struct EnsureUnique {
  using N = typename Loop<S, Sigma, Sigma>::type;
  using type = typename If<
    Equal<N, S>::value,
    Pair<S, Sigma>,
    Pair<N, Cons<Pair<S, Var<N>>, Sigma>>
  >::type;
};
```

Evaluation

The primary template takes an environment and a term

```
template<typename Sigma, typename T>
struct Eval {};
```

and is the one we'll be specializing for variables, abstractions and applications.

Let's start with variables. We look up the term corresponding to the name, and recursively evaluate this term. In case the term is a variable we stop recurring. This condition is handled by `EvalVarHelper`, which conditionally instantiate Eval recursively.

```
template<typename V, typename T, typename Sigma>
struct EvalVarHelper {
  using type = typename Eval<Sigma, T>::type;
};

template<typename V, typename Sigma>
struct EvalVarHelper<V, Var<V>, Sigma> {
  using type = Var<V>;
};

template<typename Sigma, typename V>
```

```
struct Eval<Sigma, Var<V>> {
  using T = typename Assoc<V, Sigma>::type;
  using type = typename EvalVarHelper<V, T, Sigma>::type;
};
```

Evaluating abstractions is the easiest part. We generate a unique name, insert it into the environment and recursively evaluate the term in the abstraction.

```
template<typename Sigma, typename X, typename T>
struct Eval<Sigma, Abs<X, T>> {
  using X2Sigma = typename EnsureUnique<X, Sigma>::type;
  using X2 = typename X2Sigma::First;
  using Sigma2 = typename X2Sigma::Second;
  using T2 = typename Eval<Cons<Pair<X2, Var<X2>>, Sigma2>, T>::type;
  using type = Abs<X2, T2>;
};
```

Finally we have application. We recursively evaluate both terms, and if the left hand side evaluates to an abstraction we evaluate this by inserting a binding mapping the name of the variable to the right term. This is done as follows:

```
template<typename Sigma, typename T1, typename T2>
struct EvalAppHelper {
  using type = App<T1, T2>;
};

template<typename Sigma, typename X, typename T, typename T2>
struct EvalAppHelper<Sigma, App<X, T>, T2> {
  using X2Sigma = typename EnsureUnique<X, Sigma>::type;
  using X2 = typename X2Sigma::First;
  using Sigma2 = typename X2Sigma::Second;
  using type = typename Eval<Cons<Pair<X2, T2>, Sigma2>, T>::type;
};

template<typename Sigma, typename T1, typename T2>
struct Eval<Sigma, App<T1, T2>> {
  using T1E = typename Eval<Sigma, T1>::type;
  using T2E = typename Eval<Sigma, T2>::type;
  using type = typename EvalAppHelper<Sigma, T1E, T2E>::type;
};
```

And that's our interpreter! Let's test it. We define true, false and conjunction in lambda calculus as follows:

$$T = \lambda t . \lambda f . t$$
$$F = \lambda t . \lambda f . f$$

(Ab)using the integer type from previously we can translate these into C++ types:

```
using T =
  Abs<
    Int<0>,
    Abs<
      Int<1>,
      Var<Int<0>>
    >
  >;
```

```
using F =
  Abs<
    Int<0>,
    Abs<
      Int<1>,
      Var<Int<1>>
    >
  >;
```

Finally conjunction is defined as $\lambda\, a\,.\,\lambda\, b\,.\, a\, b\, F$, which is implemented as

```
using And =
  Abs<
    Int<0>,
    Abs<
      Int<1>,
      App<
        App<
          Var<Int<0>>,
          Var<Int<1>>
        >,
        F
      >
    >
>;
```

And now for the testing:

```
using R1 = Eval<Nil, App<App<And, T>, T>>::type;
static_assert(Equal<R1, T>::value, "And T T = T");

using R2 = Eval<Nil, App<App<And, F>, T>>::type;
static_assert(Equal<R2, T>::value, "And F T = F");
```

We leave the job of further testing as an exercise to the reader!