Haskell and C++ Template Metaprogramming

Bartosz Milewski

Why Haskell?

- Easy syntax
- Almost one-to-one match with C++ TMP
- Differences
  - Runtime vs. compile-time
  - Regular data vs. types

Plan

- Functions and Metafunctions
- Lists
- Higher-Order Functions
- Closures
- Variadic Templates and TPPs
- List Comprehension
- Continuations
- Alternative Paradigms
- Bibliography

Teaser

```
template<template<class...> class cont,
                template<class> class f,
                class... lst>
struct map_cont {
    static const int value = cont<typename f<lst>::type ... >::value;
};
```

Functions

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

fact 4
```

```
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;
};

fact<4>::value
```

Predicates

```
is_zero 0 = True
is_zero x = False
```

```
template<class T> struct
isPtr {
    static const bool value = false;
};

template<class U> struct
isPtr<U*> {
    static const bool value = true;
};
```

Lists

```
count [] = 0
count (head:tail) = 1 + count tail
```

```
template<class... list> struct
count;

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

template<class head, class... tail> struct
count<head, tail...> {
    static const int value = 1 + count<tail...>::value;
};

int n = count<int, char, long>::value;
```

Higher-Order Functions and Closures

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

(or_combinator is_zero is_one) 2
```

```
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;
    };
};

or_combinator<isPtr, isConst>::lambda<const int>::value
```

Higher-Order Functions on Lists

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

all is_zero [0, 0, 1]
```

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

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

Continued…

```
all pred (head:tail) = (pred head) && (all pred tail)
```

```
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;
};
```

Fold Right

```
foldr f init [] = init
foldr f init (head:tail) =
    f head (foldr f init tail)
```

```
template<template<class, int> class, int, class...> struct fold_right;

template<template<class, int> class f, int init> struct
fold_right<f, init> {
    static const int value = init;
};

template<template<class, int> class f, int init, class head, class...tail>
struct
fold_right<f, init, head, tail...> {
    static const int value = f<head, fold_right<f, init, tail...>::value>::value;
};
```

Lists of Numbers

```
sum [] = 0
sum (head:tail) = head + (sum tail)
```

```
template<int...> struct sum;

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

template<int i, int... tail> struct
sum<i, tail...> {
    static const int value = i + sum<tail...>::value;
};
```

List Comprehension

```
[x * x | x <- [3, 4, 5]]
count lst = sum [1 | x <- lst]
```

```
one x = 1
count lst = sum [one x | x <- lst]
```

```
template<class T> struct
one { static const int value = 1; };

template<class... lst> struct
count {
   static const int value = sum<one<lst>::value...>::value;
};
```

Pattern Expansion

```
count lst = sum [one x | x <- lst]
```

```
template<class... lst> struct
count {
    static const int value = sum<one<lst>::value ... >::value;
};

int n = count<int, char, void*>::value;

// Expansion:
// sum<one<int>::value, one<char>::value, one<void*>::value>::value
// Not:
// sum<one<int, char, void*>::value>
// That would be:
// sum<one<lst ... >::value>
```

Map

```
map f lst = [f x | x <- lst]
```

```
// Does not compile!

template<template<class> class f, class... lst> struct
map {
    typedef f<lst>... type;
};
```

Continuations

```
map_cont cont f lst = cont [f x | x <- lst]
```

```
count_cont lst = map_cont sum one lst
```

```
template<template<class...> class cont,
         template<class> class f,
         class... lst>
struct
map_cont {
  static const int value = cont<typename f<lst>::type ... >::value;
};
```

Alternative Paradigms

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

```
metacode all (Pred, T...) {
    foreach (t; T)
        if (!Pred(t))
            return false;
    return true;
}
```

```
template allSatisfy(alias F, T...) {
    static if (T.length == 1)
        alias F!(T[0]) allSatisfy;
    else
        enum bool allSatisfy = F!(T[0]) && allSatisfy!(F, T[1 .. $]);
}
```

Conclusions

- C++ Template Metaprogramming = Subset of Haskell + Maximally Obfuscated Syntax
- C++ TMP not designed but discovered
- Functional paradigm just a fluke.
- Everything could be done using imperative paradigm (see Daveed Vandevoorde's proposal)

Bibliography

- http://BartoszMilewski.wordpress.com contains the blog version of this talk
- Andrei Alexandrescu, Modern C++ Design
- David Abrahams and Aleksey Gurtvoy, C++ Template Metaprogramming
- Variadic Templates, Douglas Gregor, Jaakko Järvi, and Gary Powell
- Daveed Vandevoorde, Reflective Metaprogramming in C++.