Intrinsic Currying for C++ Template Metaprograms

Symposium on Trends in Functional Programming 2018

Paul Keir [1] Andrew Gozillon [1] Seyed Hossein HAERI [2]

[1]School of Engineering and Computing
University of the West of Scotland, Paisley, UK

[2]ICTEAM Institute
Université catholique de Louvain, Louvain-la-Neuve, Belgium

June 11th, 2018

Overview

- C++ Template Metaprogramming
- Notably Absent Functional Programming Features
- The Benefits of Currying
- The Curtains Metaprogramming Library
- Interesting Observations
- Related Work
- Conclusions and Future Work

A Pure Functional Language

- C++ templates are Turing Complete
- Originally intended to allow generic function definitions
- All calculations are performed at compile time
- Types are the result, but the types themselves are untyped
- Often referred to as metaprogramming (TMP)
  - ...so too involving metafunctions, metavalues and metaexpressions
- The language is pure - no IO beyond error messages

```
template <class T> T add(T x, T y) { return x+y; }
```

A Pure Functional Language

- C++ templates are Turing Complete
- Originally intended to allow generic function definitions
- All calculations are performed at compile time
- Types are the result, but the types themselves are untyped
- Often referred to as metaprogramming (TMP)
  - ...so too involving metafunctions, metavalues and metaexpressions
- The language is pure - no IO beyond error messages

```
template <class T> T add(T x, T y) { return x+y; }
```

```
template <class T, class ...Ts> struct Foo { using type = T; };
using f_t = Foo<int,double,char**>::type;
```

Missing Features

- So, a pure functional language
- C++ standard library support; e.g. "type traits"
- ...but we would like a little more:

Missing Features

- So, a pure functional language
- C++ standard library support; e.g. "type traits"
- ...but we would like a little more:

Shopping List:

- Higher Order Functions
- Currying
- Operators
- Lambda Functions
- Type Checking
- Type Inference
- Laziness
- Type Classes

Higher Order Functions (HOFs) and Currying

- HOFs can be achieved
  - without standard library support;
  - using idiomatic TMP conventions
- Naive metafunction application, simply "returns" itself
  - e.g. *(id 43)* returns *(id 43)*
- First order metavalues can be extracted ad-hoc...
  - and used in any (type) expression: *let n = getValue $ id 43*
- But naïve higher-order metafunctions are not types...
  - by analogy: ~~*let f = getValue $ id*~~
- We can at least wrap metafunctions
  - So allowing, say: *let f = quote id*
- Combinators such as *invoke* expect wrapped metafunctions:

```
$ invoke (quote id) 43
43
```

Currying

With the simple *invoke* and *quote*, we can support HOFs:

```
$ let id' = invoke (quote id) (quote id)
$ invoke id' 43
43
```

- But *invoke* with a curried expression will fail: ~~*invoke (quote id)*~~
- Here, *quote id* (and so *id*) [1] [2] expects a single argument
  - ...and so too the failing: ~~*invoke (quote id) (quote id) 43*~~
- We now find the lack of currying a significant obstacle

[1] Hereafter, assume metafunctions have already been wrapped using *quote*
[2] As such, they are referred to as metafunction classes (MFCs)

Intrinsic Currying

- Function application in Haskell is written $e1\ e2$
  - ...where $e2$ is an arbitrary expression; and
  - $e1$ is an expression with a function type.
- Application associates to the left
- So the parentheses may be omitted in $(f\ x)\ y$
- Function application is implicitly curried

Intrinsic Currying

- Function application in Haskell is written *e1 e2*
  - ...where *e2* is an arbitrary expression; and
  - *e1* is an expression with a function type.
- Application associates to the left
- So the parentheses may be omitted in *(f x) y*
- Function application is implicitly curried
- We seek a metafunction evaluator `eval<e1,e2[,...]>`
- Ellipsis represents an optional trailing list of type arguments
- Metafunction application should also associate to the left
- Hence `eval<eval<F,X>,Y>` could be denoted as `eval<F,X,Y>`

Code Re-use and Functional Programming

- Common (type) lists are a basic but powerful data structure
- HOFs such as *map* and *fold* can create many list functions:[3]

```
> let sum            = foldr (+) 0
> let length         = foldr (\x n -> 1 + n) 0
> let reverse        = foldr (\x xs -> xs ++ [x]) []
> let map f          = foldr (\x xs -> f x : xs)  []
> let foldl f v xs   = foldr (\x g -> (\a -> g (f a x))) id xs v
> let scanr f z      = foldr (hcons f) [z]
|                        where hcons g x xss = (x `g` head xss) : xss
```

[3] See Hutton, G. "A tutorial on the universality and expressiveness of fold" (1999)

Code Re-use and Functional Programming

- Common (type) lists are a basic but powerful data structure
- HOFs such as *map* and *fold* can create many list functions:[3]

```
> let sum            = foldr (+) 0
> let length         = foldr (\x n -> 1 + n) 0
> let reverse        = foldr (\x xs -> xs ++ [x]) []
> let map f          = foldr (\x xs -> f x : xs)  []
> let foldl f v xs   = foldr (\x g -> (\a -> g (f a x))) id xs v
> let scanr f z      = foldr (hcons f) [z]
|                       where hcons g x xss = (x `g` head xss) : xss
```

- Note the subtle and intrinsic currying used above
  - The *f* argument to *map* need not be unary
    (the *map* result may be a list of functions)
  - The use of *foldr* in *foldl* is given *four* arguments
  - The *hcons* function application in *scanr* is clearly curried
- Even simple expressions such as *(foldr id 43 [id])*
  ...expect curried evaluation of *(id id 43)*
  ...which, as before, will fail when evaluated using *invoke*

[3] See Hutton, G. "A tutorial on the universality and expressiveness of fold" (1999)

Reflecting on Aims

- Without implicit currying, we cannot build on FP algorithms
- We require an evaluation mechanism, but *invoke* is too weak
- Our aim is to build the evaluator itself using a (bootstrap) fold
- Targeting a concise, trusted, verified kernel
- Let the fold guide us past corner cases
- The left fold below will drive all our currying evaluators
- Idiomatically variadic; private; implementation level API:

```
template <class, class Z, class...>
struct ifoldl
{ using type = Z; };

template <class F, class Z, class T, class... Ts>
struct ifoldl<F,Z,T,Ts...>
{ using type = typename ifoldl<F,invoke<F,Z,T>,Ts...>::type; };
```

What binary combining operation will produce the evaluator?

3 Different Implicitly Currying Left-Folding Evaluators

1. Method 1: Classic
   - Metafunctions with a single, intrinsic non-zero arity
   - Positive alignment with Haskell/OCaml norms
   - The simplest implementation: 30 lines
2. Method 2: Variadic
   - Metafunctions with one or more valid arities, including zero
   - Accommodates idiomatic nullary & variadic metafunctions
   - Explicit, incremental type-check of each additional argument
   - Albeit a heuristic search; stops (SFINAE) before the first failure
3. Method 3: Numeric
   - Metafunctions with a single, explicit numeric arity
   - A metafunction's arity is reduced by one with each argument
   - A step towards type-checking, but insufficient alone:
     - Arity of $(const\ ::\ a -> b -> a)$?
     - Count the arrows outwith parentheses; $const$ has arity 2
     - But the arity of $(const\ x)$ depends on x
     - $(const\ id)$ has arity 2; $(const\ const)$ has arity 3
   - This scheme only works as all functions can have arity of 1

Method 1: Invocation with Conditional Currying

Precondition: f is a possibly curried metafunction class
Precondition: t is an arbitrary type
Postcondition: g is either a type, or curried metafunction class

```
function CURRY-INVOKE(f, t)
    if ISVALIDEXPRESSION(f(t)) then
        g ← f(t)
    else
        g ← CURRY(f, t)
    end if
    return g
end function
```

Method 2: Heuristic, Recursive Invocation

Precondition: f is a possibly curried metafunction class
Precondition: t is an arbitrary type
Postcondition: g is a curried metafunction class

function $\textsc{Curry-invoke-peek}(f, t)$
    if $\textsc{IsValidExpression}(f()) \wedge \neg\textsc{IsValidExpression}(f(t))$ **then**
        $f' \leftarrow f()$
        $g \leftarrow \textsc{Curry-invoke-peek}(f', t)$
    else
        $g \leftarrow \textsc{Curry}(f, t)$
    end if
    return g
end function

Using the Curtains API

```
template <class, class, class> struct foldr_c;

template <class F, class Z>
struct foldr_c<F,Z,list<>>
{ using type = Z; };

template <class F, class Z, class T, class... Ts>
struct foldr_c<F,Z,list<T,Ts...>>
{ using type = eval<F,T,eval<foldr,F,Z,list<Ts...>>>; };

using foldr = quote_c<foldr_c>;
```

- As before, consider in Haskell: *(foldr id 43 [id])*
- This reduces to *(id id 43)* and then to *(43)*.
- Such an operation uses currying; all functions are unary
- So too `eval<foldr,id,char,list<id>>` $\equiv$ **`char`**
- All fold expressions from earlier can be created similarly

Using the Curtains API

Likewise, the following simple Haskell expression:

$const\ map\ ()\ (1+)\ [0,1,2]$

Using the Curtains API

Likewise, the following simple Haskell expression:

$$const\ map\ ()\ (1+)\ [0,1,2]$$

...can now be constructed in C++ TMP using the Curtains API:

```
eval<const_,map,void,eval<add,ic<1>>,ilist<0,1,2>>
```

Defining Metafunctions using Equations

- Surprisingly a new way to define TMP HOFs becomes possible
- Using `eval`, the following nested definition seems reasonable:

```
template <class F, class G>
struct compose_t
{
  template <class T>
  using m_invoke = eval<F,eval<G,T>>;
};
```

- Nevertheless, the syntax is less than ideal; a little convoluted
- The definition is analagous to the following Haskell form:

$$(.)\ f\ g = \backslash x \mathbin{-\!\!>} f\ (g\ x)$$

- It can be convenient to also use an equational definition...

Defining Metafunctions using Equations

Currently we have:

```
template <class F, class G>
struct compose_t
{
  template <class T>
  using m_invoke = eval<F,eval<G,T>>;
};
```

```
(.) f g = \x -> f (g x)
```

Defining Metafunctions using Equations

Currently we have:

```
template <class F, class G>
struct compose_t
{
  template <class T>
  using m_invoke = eval<F,eval<G,T>>;
};
```

$$(.)\ f\ g = \backslash x \rightarrow f\ (g\ x)$$

Now we can use:

```
template <class F, class G, class T>
using compose_t = eval<F,eval<G,T>>;
```

...which is comparable to the equational definition of compose:

$$(.)\ f\ g\ x = f\ (g\ x)$$

Testing Compose

```
template <class F, class G, class T>
using compose_t = eval<F,eval<G,T>>;
```

- Let's test a composition involving non-unary metafunctions
- Consider Haskell's *((.) const id 1 2)*
- ...and Curtains' `eval<compose,const_,id,int,char>`
- As expected, they reduce to *1* and `int` respectively

The Strict Fixed-point Combinator

- Laziness allows Haskell a concise fixed-point combinator:

$$fix\ f = f\ (fix\ f)$$

- Languages with eager evaluation, can use an η-expanded form
- This form is known as the Z combinator (OCaml):

$$let\ rec\ fix\ f\ x = f\ (fix\ f)\ x;;$$

The Strict Fixed-point Combinator

- Laziness allows Haskell a concise fixed-point combinator:

$$fix\ f = f\ (fix\ f)$$

- Languages with eager evaluation, can use an η-expanded form
- This form is known as the Z combinator (OCaml):

$$let\ rec\ fix\ f\ x = f\ (fix\ f)\ x;;$$

- The Curtains definition of `fix` is isomorphic:

```
template <class,class> struct fix_c;

using fix = quote<fix_c>;

template <class F, class X>
struct fix_c { using type = eval<F,eval<fix,F>,X>>; };
```

Related Work

- A. Sinkovics & Z. Porkoláb (2009): *"Expressing C++ Template Metaprograms as Lambda Expressions"*
- A. Sinkovics (2011) *"Nested Lambda Expressions with Let Expressions in C++ Template Metaprograms"*
- Louis Dionne (2013) *"Hana"*
- Eric Niebler (2014) *"Meta"*
- Peter Dimov (2018) *"Adding support for type-based metaprogramming to the standard library"* P0949R0

Conclusion and Future Work

- Curtains: a TMP library for intrinsic currying
  - Equational definition for higher order metafunctions
  - Supports nullary and variadic metafunctions
  - Check out the code on Bitbucket: `https://bitbucket.org/pgk/curtains`
  - Further folds are defined there; and in the TFP paper

Future work will target:

- Laziness
- Infix Operators
- Type Checking - perhaps via C++ Concepts
- Algebraic Data Types
- Type Classes