A Calculus for Reflective Metaprogramming

Weiyu Miao   Jeremy Siek

University of Colorado at Boulder

Outline

1. C++ Templates
   1. supportive of reflective metaprogramming
   2. drawbacks concerning syntax, efficiency, and type-checking.
2. MetaML and Garcia's Calculus for metaprogramming
3. A calculus for reflective metaprogramming

Metaprogramming Overview

Metaprogramming : the writing of programs that write or manipulate other programs.

Meta language : the language in which the metaprogram is written

Object language : the language that are manipulated.

Reflection : the ability of a programming language to be its own meta language

Well-known Application : C++ templates

C++ Templates Support Code Generation

```
// m^N = power(m, N)
// m^N --generate--> m * ... * m
template<int N>
int powN(int m) { return m * powN<N−1>(m); }

template<>
int powN<0>(int m) { return 1; }

template int powN<3>(int);     // m * m * m * 1

/*
template<> int powN<3>(int m) { return m * powN<2>(m); }
template<> int powN<2>(int m) { return m * powN<1>(m); }
template<> int powN<1>(int m) { return m * powN<0>(m); }
*/
```

Drawback: C++ templates do not support the generation of stand-alone expressions.

C++ Templates Support Static Computation

```
// M^N
template<int M, int N>
struct powMN {
  static const int value = M * powMN<M, N−1>::value;
};

template<int M>
struct powMN<M, 0> {
  static const int value = 1;
};

// 2^3 = 8 computed at compile time
int pow_2_3 = powMN<2, 3>::value;
```

Drawback: C++ templates are inefficient in memory space.

C++ Templates Support Type Reflection

// In Boost Library: libs/type_traits
is_array<T>, is_class<T>, is_pointer<T>, ...

// type tranformation
remove_const<T>, remove_pointer<T>, ...

Drawback: Inadequate reflection on types. (e.g. cannot iterate over methods and attributes in a class type)

C++ Templates Are Type-checked at Instantiation Time

In C++, type errors residing in templates are reported at instantiation time.

C++ Templates do not support a modular type system.

Example One:

```
template<typename T>
T Double (T x) { return 2 * x; }

int a[3] = {1, 2, 3};
Double(a);
// error: invalid operands of types 'int' and 'int*' to binary operator*
```

Example Two:

```
list<int> l;
std::sort(l.begin(), l.end());
```

```
sort_list.cpp:8: error: no matching function for call to sort(std::_List_iterator<int>, std::_List_iterator<int>)'
.../stl_algo.h:2835: note: candidates are: void std::sort(_Iter, _Iter) [with _Iter = std::_List_iterator<int>]
sort_list.cpp:8: note: no concept map for requirement std::MutableRandomAccessIterator<std::_List_iterator<int> >'
...
```

MetaML's Syntax for Staged Computation

meta expression	meta eval.	object-level eval.
$1 + 2$	3	3
$< 1 + 2 >$	$< 1 + 2 >$	3
$(< 1 + 2 >, 1 + 2)$	$(< 1 + 2 >, 3)$	$(3, 3)$
if false then $< 1 >$ else $< 1 + 2 >$	$< 1 + 2 >$	3
$< 1+ \sim< 2 + 3 >>$	$< 1 + 2 + 3 >$	6

MetaML-style Meta Computation and Code Generation

```
// meta computation
letrec meta powMN =
  fun(M : int, N : int) ⇒
    if N = 0 then 1 else M ∗ powMN(M, N−1);

// code generation
// powN_aux : code int → int → code int
letrec meta powN_aux =
  fun(M : (code int), N : int) ⇒
    if N = 0 then <1> else <~ M ∗ ~ powN_aux(M, N−1)>;

let meta powN =
  fun(N : int) ⇒ <fun(M : int) ⇒ ~ powN_aux(<M>, N)>;
// powN(3) = <fun(M : int) ⇒ M ∗ M ∗ M ∗ 1>
```

Types are part of the meta language and they can be manipulated as ordinary meta data.

typeof(x) == **typeof**(y); **if** x **then int else bool**; $\rightarrow\ ^{?}$(T); **dom**(**int**→ **bool**); ...

letrec meta ty2str =
  fun(x : **type**) ⇒ **match** x **with**
    | **int** → "int"
    | **bool** → "bool"
    | (x1 → x2) → ty2str(x1) + " → " + ty2str(x2)
    | _ → "unknown type";

Typechecking (MetaML Generates Well-typed Code)

MetaML : programs are fully type-checked before meta-evaluation.

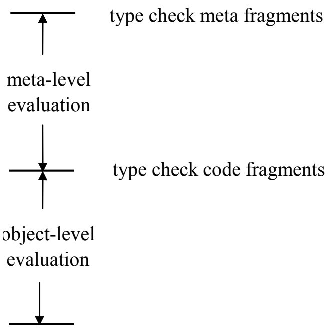
Garcia's Calculus : code fragments are type-checked after meta-evaluation.

Issue : Type Safety vs. Expressiveness

// In Garcia's calculus, cond : code
// In MetaML, cond : code int
let meta cond =
  if true then <1> **else** <2>
in <**if** ~ cond **then** 2 **else** 3>;

$\xrightarrow{meta-eval}$

<**if** 1 **then** 2 **else** 3> // not well−typed



Typechecking (Garcia's Calculus Generates More Expressive Code)

```
// C−like print function : printf(”Error: %s on line %d”, msg, line)

type Format = D | S | L of string;
// trans : string → Format
// trans(”%d”) returns D.
// trans(”Error”) returns L(”Error”).
// trans(%s) returns S.

// In Garcia's Calculus, generic_print : Format → code
letrec meta generic_print =
  fun(x : Format) ⇒ match x with
    | S → <fun(s : string) ⇒ s>              // code (string → string)
    | D → <fun(d : int) ⇒ toString(d)>      // code (int → string)
    | (L s) → <s>;                           // code string

// for MetaML, the types for each branch cannot be unified.
```

- Type-check code fragments prior to meta-evaluation. (dependent types)
- Extend Garcia's calculus with support for class reflection. (record type)
- Maintain Expressiveness. (generalized algebraic data types)

Type Checking

1. Type-check code pieces before meta evaluation, like MetaML

```
let meta cond =
  if true then <1> else <2>
in <if ~ cond then 2 else 3>;
// cond : code int; <if ~ cond then 2 else 3> not well−typed

let meta Double =
  fun(t : type) ⇒
    <fun(x : t) ⇒ 2 ∗ x>;
// t ≠ int
// <fun(x : t) ⇒ 2 ∗ x> not well−typed
```

2. Dependent types

```
let meta id = fun(t : type) ⇒ <fun(x : t) ⇒ x>
// id : Π t : type . code (t → t)
```

Type Checking

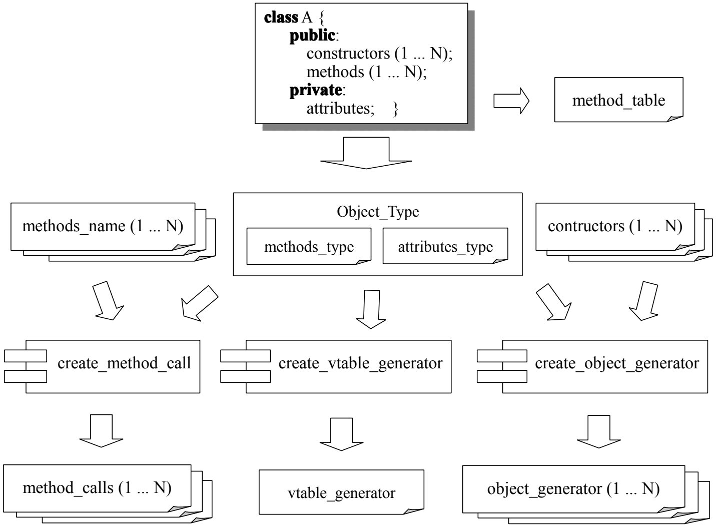
2. Dependent types support for user-defined safety checking

```
type NonEmptyList;

let meta NonEmptyListTest =
  fun(x : List) ⇒ <fun(y : if length(x)>0 then List else NonEmptyList) ⇒ y>

/*     NoneEmptyListTest :    Π x : List .
          code (if length(x)> 0 then List else NonEmptyList
                  →    if length(x)> 0 then List else NonEmptyList)      */

let meta mylist : List = [] in
  <let mytest = ~ NonEmptyListTest(mylist) in
      List_head(mytest(mylist))>

// List_head operates on a nonemptylist.
// mytest : NoneEmptyList → NoneEmptyList
// Type Error : mylist's type is not NonEmptyList.
```

Class Reflection

- simulate object-oriented features (virtual table, inheritance, overriding, etc)
- policy-based design

Function Object (I)



Function Object (II)

```
class A {
public:
   A() { count = 0; }
   A(int x) { count =x }
   int get() { return count; }
   void inc() { count = count + 1; }
private:
   int count
};
```

```
let meta attrT = { count : int ref; }    // mutable record
```

```
let meta methodT =
{ get : attrT -> void -> int, inc : attrT -> void -> void }
```

```
let meta objectT = { state : attrT, method : methodT }
```

```
let meta constructor1 =
    < fun(x : void) => { count = ref 0 }>
```

```
let meta constructor2 =
    < fun(x : int) => { count = ref x }>
```

```
let meta method_table =
    < fun(self : methodsT) =>
          {   get =
                  fun(state : attrT) =>
                     fun(x : void) => ! (state.count),
              inc =
                  fun(state : attrT) =>
                     fun(x : void) => (state.count) <- (! (state.count) + 1) } >
```

Function Object (III)

create_method_call

create_vtable_generator

create_object_generator

compile time

let vtable_generator = ~ create_vtable_generator(methodT)

let object_generator2 = ~ create_object_generator(methodT, constructor2)

let object_generator1 = ~ create_object_generator(methodT, constructor1)

let call_inc = ~ create_method_call(objectT, inc)

let call_get = ~ create_method_call(objectT, get)

run time

let vtable = vtable_generator(~ method_table)

// A a1 = **new** A();
  let a1 = object_generator1(vtable)();

// a1.get();
  call_get(a1)();

// A a2 = **new** A(1)
  let a2 = object_generator2(vtable)(1);

// a2.inc()
  call_inc(a2)();

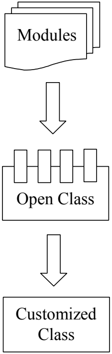
Inheritance

```
class A {
public:
   A() { count = 0; }
   A(int x) { count =x }
   int get() { return count; }
   void inc() { count = count + 1; }
private:
   int count;
};
```

```
let meta attrT = ...
let meta methodT = ...
let meta objectT = ...
let meta constructor1 = ...
let meta constructor2 = ...
```

```
class B : A {
public:
   B() { A(); }
   B(int x) { A(x); }
   void set(int x) { count = x; }
   void inc() { count = count + 10; }    //override
};
```

```
let meta method_table_B =
      < fun(self : methodT) =>
            {   set =
                   ...
                inc =       // override
                   ...  }
         | (~ method_table_A)(self) >
```




```
let meta DoNotTrace = module {
        report_head = <...> , // print null
        report_tail = <...> // print null
};

let meta DoTrace = module {
        report_head = <..., show(head)> , // print the structure of list head
        report_tail = <..., show(tail)> // print the structure of list tail
};

let meta TracePolicyType = { report_head : code (... -> string),
                             report_tail : code (... -> string)   };
```

```
let meta policy_list =
   fun(T : type, TracePolicy : TracePolicyType) =>
      ...
      <   ...
          head = ...
             fun(x : void) =>
                (~TracePolicy.report_head)(state.head), ...
          tail = ...
             fun(x : void) =>
                (~TracePolicy.report_tail)(state.tail), ...   >
```

```
let meta customized_policy_list = policy_list(int, DoNotTrace)
```

GADTs for Expressiveness

```
datatype Indexed_List : (type, int) → type where
  | Nil : Indexed_List(t, 0)
  | Cons : t → Indexed_List(t, i) → Indexed_List(t, i + 1)


datatype Format : type → type where
   | S : Format(string → string)
   | D : Format(int → string)
   | L : string → Format(string)

// generic_print : Format(t) → (code t)
letrec meta generic_print =
   fun(x : Format(t)) ⇒
       match x with
          | S → <fun(x : string) ⇒ x>             // code (string → string)
          | D → <fun(d : int) ⇒ toString(d)>      // code (int → string)
          | (L s) → <s>                           // code string
       withtype (code t);
```

Related Work

- *Scheme Macros*
  staged computation, type-check after macro expansion, no reflection over types
- Tim Sheard and Simon Jones. *Template Haskell*
  two-staged computation, type-check during meta-evaluation, little reflection over types, GADTs
- Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. *Concoqtion*
  staged computation, dependent types, decidable type-checking, reflection over types, GADTs

Conclusion

In this talk, we extend Garcia's calculus with several more features needed to provide the full power of C++ templates, including reflection over classes and user-defined safety checks. In addition, we present a modular type system that catches errors at definition time.