Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms*

Ádám Sipos, Zoltán Porkoláb, Norbert Pataki, Viktória Zsók

{shp|gsd|patakino|zsv}@elte.hu

Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University, Budapest

* Supported by GVOP-3.2.2.-2004-07-0005/3.0 and SA 66ÖU2
Contents

• Template Metaprogramming
• Metaprogramming & Functional Programming
• A Functional TMP interface
• Example: Lazy data types
• Evaluation
C++ Template Metaprogramming I

• Template:
  – `list<int>, list<double>`
  – Tool for generic programming
  – Unconstrained (in current standard)
  – Concepts (in C0x)

• Implementation
  – Instantiation (in compile-time)
  – Specialization
  – Pattern-matching

• Compile-time algorithms
  – Recursions, conditional statements => Turing-completeness
C++ TMP: an example

template <int N>
struct Factorial
{
    enum { value = N*Factorial<N-1>::value };`n`n
template <>
struct Factorial<1>
{
    enum { value = 1 };`n`n

int main()
{
    int r = Factorial<5>::value;
}
Application areas of TMP

- Concept checking
- Expression templates
- Compile-time code adaptation
- Compiler construction
- Active libraries
- Language embedding
- etc
TMP is Functional

• Referential transparency
  – no mutable variables
  – no loops
• Pattern matching
• Higher order TMPs
• Lazyness?
Motivation

• Terrible syntax
• Terrible diagnostics
• Metaprogramming in itself is difficult and ugly
• Most metaprograms are handcrafted
• Non-functional style libraries (like boost::mpl)
• Hard to debug
• Hard to predict the development process

• Aim: C++ template metaprogramming interface which explicitly expresses the programmer’s intentions
Meta<Fun>

- Aim: C++ template metaprogramming interface which explicitly expresses the programmer’s intentions

- Embedded functional-style interface for C++ Template Metaprograms

- Embedded language example: Clean
  - Pure functional language based on graph rewriting with outermost tree reduction
  - Lazy evaluation
Structure of Meta<Fun>
Components of Meta<Fun>:

- Clean-TMP translator
  - Work in progress

- Clean-like metaprogram library
  - Separating the user-written code from the graph-rewriting engine
  - Equivalent semantics to Clean programs
  - Library based on standard C++
  - Portable
An example: Sieve

take 0 xs = []
take n [x:xs] = [x:take (n-1) xs]

sieve [prime:rest] = [prime : sieve (filter prime rest)]
filter p [h:tl] | h rem p == 0 = filter p tl
= [h : filter p tl]
filter p [] = []

Start = take 10 (sieve [2..]) ->
take 10 ([2, sieve (filter 2 [3..])]) ->
[2, take 9 (sieve (filter 2 [3..]))] ->
[2, take 9 (sieve [3, filter 2 [4..]])] ->
[2, take 9 [3, sieve (filter 3 (filter 2 [4..]))]] ->
[2, 3, take 8 (sieve (filter 3 (filter 2 [4..])))] ->
..........
Sieve: translated code

template <int p, int x, class xs>
struct filter<p, Cons<x,xs> >
{
    typedef typename
        if_c< x%p==0, Filter<p,xs>, Cons<x,filter<p,xs> > >::type right;
};

template <class xs>
struct sieve
{
    typedef NoMatch right;
};

template <int p, class xs>
struct sieve<Cons<p,xs> >
{
    typedef Cons<p,sieve$filter<p,xs> > > right;
};
Lazy list and Sieve

template <int r>
struct EnumFrom
{
    typedef Cons<r, EnumFrom<r+1> > right;
};

take<10, sieve<EnumFrom<2> >
take<10, Cons<2, sieve<filter<2, EnumFrom<3> > > >
Cons<2, take<9, sieve<filter<2, EnumFrom<3> > > >
Cons<2, take<9, sieve<3, filter<2, EnumFrom<4> > > >
Cons<2, take<9, Cons<3, sieve<filter<3, EnumFrom<4> > > >
Cons<2, 3, take<8, filter<3, filter<2, EnumFrom<4> > > >
............
The engine

template <class T1, template <class> class Expr>
struct Eval<Expr<T1> >
{

    typedef typename 
        if_c<is_same<typename Expr<T1>::right,NoMatch>::value,
            typename 
                if_c<!Eval<T1>::second,
                    Expr<T1>,
                    Expr<typename Eval<T1>::result>
                >::type,
            typename Expr<T1>::right
        >::type result;

    static const bool second =
        !(is_same<typename Expr<T1>::right,NoMatch>::value &&
          !Eval<T1>::second);
};
Compilation times
Related/Future work

• C++ Template Metaprogram Libraries
  – Loki (Alexandrescu 2000)
  – boost::mpl (Gurtovoy, Abrahams, 2002)
    • Incl: TMP Lambda
• Runtime functional library
  – fc++ (McNamara, Smaragdakis, 2000)
• Func – C++ template mapping
  – From Haskell type classes to C++ concepts
    (Zalewski, Priesnitz, Ionescu, Botta, Schupp, 2007)

• Closer to Clean syntax
• Order of rules (priorities)
• Optimizations
Conclusion

• C++ Template Metaprogramming is emerging
• Functional-style programming
• Current syntax is unsuitable and source of errors
• Functional syntax would better express the programmers’ intentions
• Meta<Fun> - a portable way to achieve the goals
• Lazy data types – a proof of concepts
• Many open questions
BEGIN_CLEAN(sievesum10)

take 0 xs = []
take n [x:xs] = [x:take (n-1) xs]

sieve [prime:rest] = [prime : sieve (filter prime rest)]
filter p [h:tl] | h rem p == 0 = filter p tl
               = [h : filter p tl]
filter p [] = []

Start = sum ( take 10 (sieve [2..]) )

END_CLEAN

static const int prime10 = cleanstart(sievesum10);
Thank you!
Questions?

Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms

Ádám Sipos, Zoltán Porkoláb, Norbert Pataki, Viktória Zsók

{shp|gsd|patakino|zsv}@elte.hu

Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University, Budapest